Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Paralelização do Algoritmo de Indexação de Dados Multimídia Baseado em Quantização

André Fernandes Freire

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. George Luiz Medeiros Teodoro

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Paralelização do Algoritmo de Indexação de Dados Multimídia Baseado em Quantização

André Fernandes Freire

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. George Luiz Medeiros Teodoro (Orientador)
CIC/UnB

Prof. Dr. Alba Cristina Magalhaes Alves de Melo
CIC/UnB

Prof. Dr. Edison Ishikawa
CIC/UnB

Prof. Dr. Edison Ishikawa
Coordenador do Bacharelado em Ciência da Computação

Brasília, 30 de janeiro de 2019

Agradecimentos

Sou grato a Deus e a meu salvador Jesus Cristo, por dar sentido a tudo que faço. Entender a ciência apenas como uma forma de investigar a natureza ou de resolver problemas e desenvolver novas tecnologias carece de sentido. Será que vale a pena saciar aos poucos nossa curiosidade sobre a natureza? Será que o avanço tecnológico realmente tem resolvido os problemas da humanidade? Para mim, tudo isso é periférico se no fim todos formos condenados à inexistência. Meus esforços em produzir algo vem de fora de mim, do entendimento que investigar a natureza é conhecer Deus através de sua criação, da certeza que minha capacidade de desenvolver é um dom que deve ser usado pra glorificar o Criador, da fé de que tudo faz parte de um plano maior e eterno. Se há algum mérito nesse trabalho, é d'Aquele que me deu a capacidade, os meios e as ferramentas para realizá-lo.

Agradeço também a minha família. Aos meus irmãos, Bruno e Felipe, e a minha mãe, Jeanette, que se dedicaram a mim desde o início. Não mediram esforços quando, no ensino fundamental, eu tive a primeira oportunidade de ter contato com a universidade. Abandonaram seus fins de semana para me levar de ônibus até outra cidade, onde eu teria minhas aulas de iniciação científica. E mais tarde, me deram todo o apoio e os meios para que eu viesse estudar na UnB, mesmo que isso significasse ficar distante.

Deixo minha eterna gratidão a todos aqueles que me apoiaram nessa caminhada. Ao George, que me incluiu nesse projeto e me orientou em tudo que precisei para sua realização. Aos meus amigos da mecatrônica, que não me abandonaram mesmo depois de eu abandoná-los. Aos meus amigos mais antigos, que mesmo estando nas mais diferentes cidades continuaram do meu lado. A minha igreja, que me deu suporte emocional e espiritual sempre que precisei. E à Universidade de Brasilia e à CAPES, por investirem nesse projeto.

Resumo

A busca por similaridade em espaços de alta dimensionalidade é uma operação fundamental em diversas aplicações de recuperação de dados multimídia, no entanto essa operação é tipicamente uma das mais computacionalmente caras. Alguns métodos propõem a busca aproximada para minimizar esse problema, uma alternativa que tenta fazer um compromisso entre o custo computacional e a precisão da busca. Um dos métodos baseados em busca aproximada é o Product Quantization for Approximate Nearest Neighbor Search (PQANNS), que propõe a decomposição do espaço de busca em um produto cartesiano de subespaços de baixa dimensionalidade e a quantização de cada um deles separadamente. Para tanto, é utilizada uma estrutura de lista invertida para fazer a indexação dos dados, o que permite a realização de buscas não-exaustivas. A redução da dimensionalidade dos dados aliada à busca não-exaustiva faz com que o PQANNS responda consultas de forma eficiente e com baixa demanda de memória, no entanto sua execução sequencial ainda é limitada a trabalhar com bases que caibam na memória RAM de apenas uma máquina. Nosso objetivo é propor uma paralelização em memória distribuída do PQANNS, sendo assim capaz de lidar com grandes bases de dados. Também propomos uma paralelização em máquina multicore, visando reduzir o tempo de resposta às consultas e utilizar toda a capacidade de processamento disponível. Nossa paralelização em memória distribuída foi avaliada utilizando 128 nós/3584 núcleos de CPU, obtendo uma eficiência de 0.97 e foi capaz de realizar a indexação e busca em uma base de dados contendo 256 bilhões de vetores Scale Invariant Feature Transform (SIFT). Além disso, a execução da nossa paralelização em máquina multicore obteve um excelente ganho em desempenho com até 28 núcleos, obtendo um *speedup* médio de $26,36x$ utilizando todos os núcleos.

Palavras-chave: busca por similaridade, paralelismo, quantização

Abstract

The search for similarity in high dimensional spaces is a core operation found in several multimedia retrieval applications. However this operation is typically one of the most computationally expensive. Some methods propose an approximate search to minimize this problem, trying to make a trade-off between computational cost and search precision. One of these methods is the Product Quantization for Approximate Nearest Neighbor Search (PQANNS), which proposes the decomposition of the search space into a Cartesian product of low-dimensional subspaces and the quantization of each of them separately. In order to do so, an inverted file structure is used to index the data, which allows non-exhaustive searches. The reduction of data dimensionality coupled with the non-exhaustive search causes the PQANNS to respond efficiently and with low memory requirements, however its sequential execution is still limited to working with bases that fit into the RAM memory of a single machine. Our goal is to propose a parallelization strategy that works on distributed memory plataforms of PQANNS, thus being able to handle large databases. We also propose a multicore machine parallelization, in order to reduce the response time to the queries and to use all available processing capacity. Our distributed memory parallelization was evaluated using 128 nodes/3584 CPU cores, obtaining an efficiency of 0.97 and was able to perform the index and search in a database containing 256 billion Scale Invariant Feature Transform (SIFT) vectors. In addition, the execution of our parallelization in a multicore machine obtained a performance gain with up to 28 cores, obtaining an average speedup of $26.36x$ using all the cores.

Keywords: similarity search, parallelism, product quantization

Sumário

1 Introdução **1**
1.1 Objetivo e Contribuições 3
1.2 Organização do Texto 4

2 Referencial Teórico **5**
2.1 Métodos de Representação de Dados Multimídia 5
2.1.1 Descritores Locais 6
2.1.2 Representação por Meio de Descritores Locais 8
2.1.3 Descritores Globais 10
2.2 Recuperação de Imagens 11
2.2.1 Busca por Similaridade 11
2.2.2 Indexação e Busca Aproximada 12
2.3 Arquiteturas Paralelas 21
2.3.1 Taxonomia de Flynn 21
2.3.2 Taxonomia de Duncan 22
2.4 Soluções Distribuídas para Indexação e Busca Aproximada 24
2.4.1 Paralelizações do LSH em MapReduce 24
2.4.2 Paralelização do FLANN em MPI 25

3 Product Quantization for Approximate Nearest Neighbor Search (PQANNS) **26**
3.1 Quantização de Espaços 26
3.2 Cálculo de Distâncias em Espaços Quantizados 28
3.3 Busca Não Exaustiva em Espaços Quantizados 29
3.3.1 *Coarse Quantizer* 30
3.3.2 Lista Invertida 31
3.3.3 Busca em Espaços Quantizados 31

4 Paralelização do PQANNS **34**
4.1 Paralelização em Memória Distribuída 34
4.2 Paralelização em Máquina Multicore 38

5 Análise dos Resultados **40**
5.1 Ambiente de Desenvolvimento e Testes 40
5.2 Resultados Experimentais 40
5.2.1 Comparação com o Estado da Arte: FLANN 41
5.2.2 Avaliação de Escalabilidade em Ambientes Multicore 43
5.2.3 Avaliação de Escalabilidade em Ambientes de Memória Distribuída . . 44

6 Conclusão **46**
6.1 Trabalhos Futuros 47

Referências **48**

Lista de Figuras

1.1 Recuperação de imagens - esquema básico. 2

2.1 Solução para mudança de perspectiva e iluminação entre regiões semelhantes de imagens. Primeira linha: uma perspectiva; segunda linha: outra perspectiva. (a) Imagens originais, (b) regiões semelhantes detectadas, (c) visão mais próxima da região detectada, (d) normalização geométrica circular, (e) normalização fotométrica e geométrica.. 7

2.2 Um descritor é criado a partir da magnitude do gradiente e orientação de cada ponto amostral da imagem na região em torno do ponto de interesse, como mostrado à esquerda. Isto é ponderado por uma janela gaussiana, indicada pelo círculo. As amostras são acumuladas em histogramas de orientação sumarizando o conteúdo de regiões 4x4, como mostrado à direita, com o tamanho de cada vetor correspondendo à soma das magnitudes do gradiente próximas à direção da região. A figura ilustra um descritor 2x2 computado a partir de amostras 8x8.. 8

2.3 Ideia básica da busca por similaridade baseada em características. 12

2.4 O problema dos efeitos de borda em curvas de preenchimento de espaço. Os pontos no centro da imagem estão bem mais distantes na curva que os pontos do quadrante inferior à esquerda, mesmo que estejam mais próximos no espaço.. 20

2.5 Taxonomia de Duncan. 22

3.1 Ilustração do cálculo de distância simétrico (esquerda) e assimétrico (direita). No método simétrico, o cálculo utiliza apenas valores quantizados para determinar a distância $d(q(x), q(y))$, enquanto o assimétrico utiliza o vetor de consulta x e os valores quantizados da base de dados $d(x, q(y))$. Observe que a reta de cor vermelha corresponde à distância real, enquanto a de cor preta corresponde à distância aproximada.. 29

3.2 Visão geral do sistema de lista invertida com cálculo assimétrico de distância. 33

4.1 Decomposição do PQANNS em um paradigma de fluxo de dados. Na fase de construção do índice, os dados da base são particionados pelas cópias de Leitura da Entrada e enviados para as de Busca no Índice, sem replicação de dados (mensagem i). Nesse estágio são criadas as listas invertidas que indexam os dados locais de cada cópia. Durante a fase de busca, as cópias de Busca no Índice recebem informações sobre o vetor de consulta lido pelas de Entrada de Consultas e computa localmente os NN de sua partição da base de dados de entrada. Em seguida, o estágio de Agregação recebe os resultados locais e computa os NN globais resultantes. 36

5.1 IVFADC vs FLANN: compromissos entre qualidade da busca (precisão) e tempo de busca.. 42
5.2 Variação do *speedup* sobre o algoritmo sequencial em relação à variação do número de *threads*. 43
5.3 Escalabilidade da paralelização em memória distribuída em um experimento de escalabilidade fraca utilizando uma base de dados contendo 256 bilhões de descritores SIFT na configuração com 128 nós. 45
5.4 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. 45

Lista de Abreviaturas e Siglas

ADC Asymmetric Distance Computation.

ANN Approximate Nearest Neighbor.

BBF Best Bin First.

BLAS Basic Linear Algebra Subprograms.

BOSSA Bag Of Statistical Sampling Analysis.

BOW Bag of Words.

CPU Central Process Unit.

DHT Distributed Hash Tables.

DoG Difference of Gaussians.

EQM Erro Quadrático Médio.

FCTH Fuzzy Color and Texture Histogram.

FLANN Fast Library for Approximate Neighbors Search.

GMM Gaussian Mixture Model.

GPU Graphics Processing Unit.

HDFS Hadoop Distributed File System.

HE Hamming Embedding.

HOG Histograms of Oriented Gradients.

IVFADC Inverted File System with Asymmetric Distance Computation.

JPDC Journal of Parallel and Distributed Computing.

LAPACK Linear Algebra Package.

LoG Laplacian-of-Gaussian.

LSH Locality-Sensitive Hashing.

MIMD Multiple Instruction, Multiple Data Streams.

MISD Multiple Instruction, Single Data Stream.

MPI Message Passing Interface.

NBNN Naive-Bayes Nearest-Neighbor.

NN Nearest Neighbor.

OpenMP Open Multi-Processing.

PQANNS Product Quantization for Approximate Nearest Neighbor Search.

SDC Symmetric Distance Computation.

SIFT Scale Invariant Feature Transform.

SIMD Single Instruction, Multiple Data Streams.

SISD Single Instruction, Single Data Stream.

SURF Speeded-Up Robust Features.

VLAD Vector of Locally Aggregated Descriptors.

Capítulo 1

Introdução

O volume de dados multimídia espalhados na internet cresceu consideravelmente nos últimos anos, o que se deve principalmente à popularização de câmeras e smartphones [1, 2, 3, 4]. Em janeiro de 2016 foram consumidas 100 milhões de horas de vídeos por dia no Facebook [1]. Em 2017 o Instagram revelou que continha um total de 40 bilhões de fotos compartilhadas [2] e que no fim de 2014 já era feito o upload de 70 milhões de fotos por dia [2]. Já o Youtube ultrapassou a marca de 1 bilhão de horas de vídeo assistidas diariamente em 2017 [4].

Essa enorme massa de dados exige meios de gerenciamento que tornem o conteúdo acessível. Um dos meios é permitir a busca por meio do conteúdo dos dados [5], que será o foco deste trabalho. A ideia básica é extrair características que descrevem os dados ou objetos, mapeá-las em vetores de alta dimensionalidade e realizar a busca de objetos com vetores de características similares na base de dados. Cabe notar que a busca é um passo fundamental e tipicamente um dos mais caros computacionalmente.

O escopo desse tipo de aplicação é muito amplo, abrangendo motores de busca de imagens, reconhecimento de imagens em dispositivos móveis, identificação de músicas em tempo real e reconhecimento de material com direitos autorais. Além disso, uma melhor manipulação de dados multímidia pode contribuir no avanço da inteligência artificial, da interação humano-robô e da direção assistida, dentre outras áreas [6].

Para realizar a busca baseada em conteúdo em uma base de dados multimídia é necessário calcular a similaridade entre um determinado dado de consulta e os dados da base, retornando os mais similares. Por exemplo, dada uma imagem de consulta a ser buscada em um banco de imagens, um algoritmo calcula o nível de similaridade entre a imagem dada e as imagens da base e retorna uma lista com as mais semelhantes (veja Figura 1.1).

É importante que esse tipo de aplicação utilize representações fiéis do conteúdo, o que significa que a representação deve superar a disparidade entre a linguagem baixo nível do computador e o significado alto nível de um dado multimídia. Uma solução típica é o uso

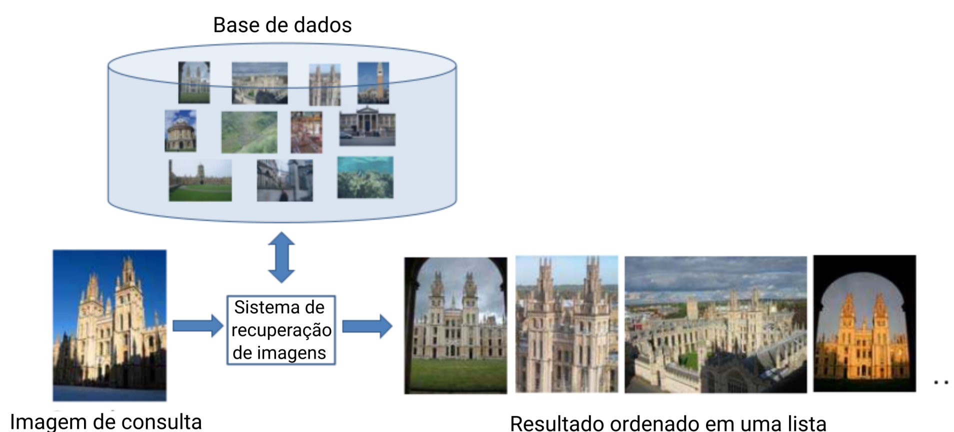

Figura 1.1: Recuperação de imagens - esquema básico (Fonte: [6]).

de descritores, que são vetores de alta dimensionalidade que descrevem o conteúdo de um dado baseado em diversas informações. Por exemplo, imagens podem ser representadas por meio de histogramas de cores [7], de descritores de formas [8, 9], vetores de Fourier [10], dentre outros.

A partir dos descritores, a similaridade entre dados multimídia pode ser calculada, por exemplo, a partir da distância euclidiana entre um descritor de consulta e os descritores da base, assim os descritores mais próximos (Nearest Neighbors - NN) são retornados na busca. No entanto realizar uma busca exaustiva é inviável devido à alta dimensionalidade dos dados e a grande quantidade de dados disponíveis para indexação. Com isso tornou-se necessária a utilização de algoritmos de indexação e estruturas de dados que reduzam o espaço de busca, por exemplo, particionando os objetos da base espacialmente. Na busca, essas partições são usadas para evitar a verificação em porções da base de dados que não contém os NN. Essas abordagens apresentaram diversos métodos de indexação, como "kd-trees" [11], "k-means trees" [12], dentre outros.

No entanto ainda há a limitação da "maldição da dimensionalidade" [13], que se refere aos diversos problemas associados à organização e análise de dados de alta dimensionalidade. Na busca por similaridade os efeitos são relacionados ao espaço de busca, pois o aumento na dimensionalidade dos dados tem um impacto exponencial no volume do espaço e na esparsidade dos dados, o que dificulta a indexação e aumenta os custos de busca. Uma forma de contornar esse problema é a busca aproximada (Approximate Nearest Neighbors - ANN), que tem como ideia central encontrar os NN com grande probabilidade de serem mais semelhantes à consulta, ao invés da busca exata. Diversos algoritmos utilizam essa abordagem, como o Fast Library for Approximate Neighbors Search (FLANN) [12], o Locality-Sensitive Hashing (LSH) [14, 15] e o Product Quantization for Approximate

Nearest Neighbor Search (PQANNS) [16].

Todos esses algoritmos levam em consideração o desempenho na execução e a qualidade da busca, no entanto é inevitável que a priorização de um desses fatores comprometa o outro. Uma solução eficiente deve conseguir lidar com as prioridades de diferentes aplicações comprometendo o mínimo possível o outro fator. Em aplicações médicas a fidelidade da representação é mais importante que desempenho, enquanto motores de busca podem trabalhar com representações menos fiéis, desde que a busca seja realizada rapidamente e use poucos recursos.

Apesar de algumas propostas fornecerem bons compromissos entre os fatores citados, sua implementação sequencial não consegue atender aplicações que trabalham com grandes bases de dados, que tipicamente excedem a memória de uma única máquina. Por isso é importante o uso de uma arquitetura que permita o processamento paralelo das consultas, com a utilização de diversas máquinas. Além disso, processadores modernos contam com diversos núcleos de processamente em um chip, o que torna interessante uma implementação que faça uso de todos os núcleos de forma concorrente.

Alguns algoritmos foram propostos nesse contexto, como as paralelizações do LSH utilizando MapReduce [17, 18], do FLANN [19] em cima do Message Passing Interface (MPI) e do PQANNS [20, 21] para GPU (microprocessador composto por uma robusta arquitetura paralela capaz de lidar com múltiplas tarefas simultaneamente), mas todos sofrem com alguma limitação de escalabilidade. As implementações do LSH em MapReduce não resolvem o problema de construir um índice de busca de larga escala que preserve o comportamento do algoritmo sequencial. A paralelização do FLANN é limitada pela largura da banda de memória e as implementações do PQANNS em GPU são limitadas a uma máquina e não conseguem lidar com uma grande quantidade de dados.

O PQANNS obteve compromissos melhores que os outros algoritmos, obtendo melhores tempos de execução e menor uso de memória para uma mesma qualidade de busca. No entanto suas implementações focaram em obter melhor desempenho em uma execução sequencial e até mesmo suas paralelizações [20, 21] lidam apenas com uma máquina. Com isso vimos a necessidade de propor uma implementação em um ambiente de memória distribuída que contemple a grande quantidade de dados disponíveis para indexação.

1.1 Objetivo e Contribuições

O objetivo deste trabalho é propor uma solução para o tratamento de grandes bases de dados multimídia na busca por similaridade, através da paralelização em memória distribuída do algoritmo PQANNS. A partir disso, pudemos trazer as seguintes contribuições para o tema:

- Projetamos e implementamos uma paralelização em memória distribuída do algoritmo PQANNS, que lida com a indexação de grandes bases de dados enquanto mantém um bom compromisso entre o tempo de execução, precisão da busca e uso de memória.
- Apresentamos testes comparativos com um dos principais algoritmos do estado da arte, o FLANN, que mostram o baixo uso de memória do PQANNS em comparação à busca do FLANN com tempos de execução e precisão próximos.
- Mostramos a capacidade de escalablidade da nossa paralelização através de testes utilizando uma vasta quantidade de máquinas e uma extensa base de dados.

Todas as contribuições acima foram publicadas [22], no periódico Qualis A2 *Journal of Parallel and Distributed Computing (JPDC).*

1.2 Organização do Texto

O restante do trabalho segue a seguinte organização:

- Capítulo 2 - Referencial Teórico: faz uma revisão teórica dos conceitos, métodos e técnicas apresentadas em trabalhos anteriores envolvendo métodos de representação de dados multimídia, recuperação de imagens e paralelismo;
- Capítulo 3 - Product Quantization for Approximate Nearest Neighbor Search (PQANNS): Detalha o algoritmo PQANNS;
- Capítulo 4 - Paralelização do PQANNS: Apresenta uma paralelização eficiente do PQANNS, detalhando a abordagem proposta;
- Capítulo 5 - Análise de Resultados: Descreve os resultados obtidos a partir abordagem apresentada no capítulo anterior;
- Capítulo 6 - Conclusão: Apresenta as conclusões do trabalho de graduação, discutindo os objetivos alcançados e indicando sugestões de continuação do trabalho.

Capítulo 2

Referencial Teórico

Neste capítulo serão abordados métodos de representação de imagens, apresentando os descritores locais e globais, bem como exemplos de algoritmos de extração e agregação de descritores. Também serão explorados os conceitos por trás da busca por similaridade, as principais estruturas de dados utilizadas e alguns algoritmos de indexação presentes em métodos de busca com ANN. Por fim serão apresentados alguns conceitos básicos sobre arquiteturas paralelas e abordagens distribuídas para algoritmos de indexação e busca.

2.1 Métodos de Representação de Dados Multimídia

A representação de dados multimídia é um fator importante que precede a recuperação de dados e, portanto, utilizar boas representações é importante para que o algoritmo de busca apresente resultados significativos. Nesta seção vamos focar na apresentação de diferentes métodos de representação de imagens.

O objetivo da representação visual é converter imagens em uma representação matemática onde imagens semelhantes têm representações similares e imagens diferentes têm representações diferentes. A representação deve ser robusta o suficiente para fazer essa ligação de forma precisa nos mais diferentes contextos e, ao mesmo tempo, devem ser fáceis de computar, guardar e recuperar.

Uma forma de representação é a extração de um descritor global por imagem, nesse caso apenas um vetor é necessário para caracterizar uma imagem. Apesar de ser altamente escalável, esse método é sensível a ruído de fundo, mudança de perspectiva, obstrução de informação. Para contornar esses problemas foi proposta a extração local de descritores, isto é, várias regiões são extraídas das imagens e são computados descritores para elas.

A utilização desses descritores é mais complexa, pois depende da aplicação de métodos que agreguem ou classifiquem os descritores locais para encontrar as melhores correspon-

dências. A seguir vamos detalhar a representação por meio de descritores locais e globais, apresentando os principais métodos da literatura.

2.1.1 Descritores Locais

Para realizar a extração de descritores locais é necessário detectar regiões de interesse na imagem e extrair os descritores de cada região. A extração de descritores apenas das regiões mais relevantes da imagem permite que a representação ignore ruídos presentes na imagem, bem como variações geradas por regiões de menor importância e diferenças de perspectiva.

Detecção de Regiões

Os detectores selecionam pontos de interesse nas imagens, que serão analisados em conjunto com a região que os englobam. Isso permite que a mesma região encontrada em duas imagens corresponda a uma representação bem próxima, apesar de variações de perspectiva e escala. Alguns dos detectores de regiões mais famosos são o Harris-Affine e o Hessian-Affine [23], o Difference of Gaussians (DoG) e o Laplacian-of-Gaussian (LoG) [24]. Eles podem ser caracterizados como: detector de canto, detector de *blob* e detector de região.

Métodos como o Harris-Affine e o Hessian-Affine fazem a extração de regiões por afinidade e aplicam transformações para normalizar as regiões obtidas. Isso permite a identificação de regiões semelhantes em diferentes imagens de forma em que elas correspondam à mesma imagem a partir de diferentes perspectivas. Isto é, sua forma não é fixada, mas se adapta automaticamente, baseado nas intensidades de imagens adjacentes, de forma que eles sejam a projeção da mesma superfície 3D. A Figura 2.1 apresenta um exemplo de uso de tranformações para normalizar a perspectiva e iluminação de regiões em imagens.

Outra opção é amostrar densamente pontos de características de uma grade regular ao invés extrair pontos de interesse. Essa alternativa é utilizada majoritariamente em classificação, onde quase sempre são obtidos melhores resultados por meio da amostragem densa. É algo equivalente a dar mais dados para o classificador e deixá-lo decidir o que é mais discriminativo e informativo.

Extração de Descritores

Com os pontos de interesse definidos, devem ser extraídas as características de cada ponto. Os descritores locais são os vetores que contém as informações relativas a cada região,

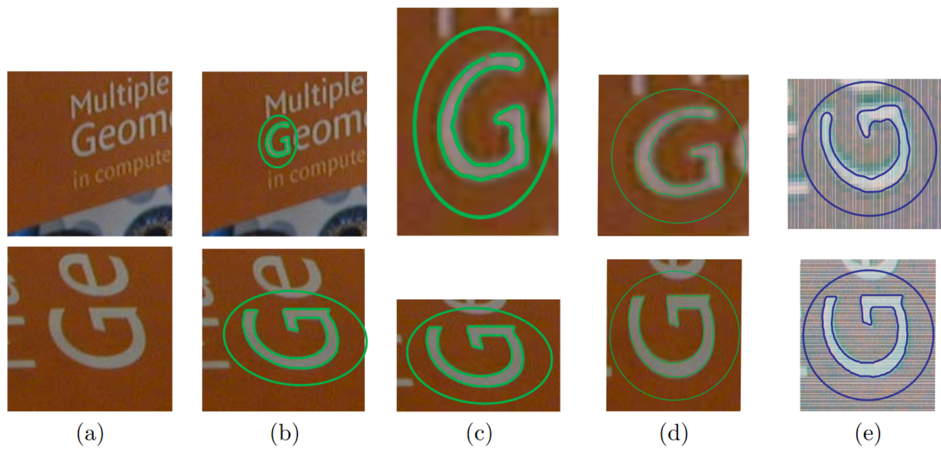

Figura 2.1: Solução para mudança de perspectiva e iluminação entre regiões semelhantes de imagens. Primeira linha: uma perspectiva; segunda linha: outra perspectiva. (a) Imagens originais, (b) regiões semelhantes detectadas, (c) visão mais próxima da região detectada, (d) normalização geométrica circular, (e) normalização fotométrica e geométrica. (Fonte: [25]).

eles devem apresentar uma representação robusta o suficiente para garantir que regiões semelhantes de diferentes imagens com transformações sejam compatíveis.

Um dos descritores de características mais famosos é o Scale Invariant Feature Transform (SIFT) [24], que extrai uma característica a partir de uma espécie de histograma do gradiente da orientação e localidade de um ponto de interesse na imagem. Além dele, outros métodos são discutidos na literatura, como sua variação Speeded-Up Robust Features (SURF) [26] e o Histograms of Oriented Gradients (HOG) [27]. No entanto vamos detalhar apenas o SIFT, por ser o principal descritor utilizado neste trabalho. De toda forma, os métodos de indexação que implementamos pode ser aplicado a qualquer desses descritores.

Nessa abordagem, um histograma de orientação é criado a partir do gradiente das orientações de pontos amostrais de uma região em torno do ponto de interesse. O histograma tem 36 posições cobrindo o alcance de 360 graus das orientações. Cada amostra adicionada ao histograma é ponderada pela magnitude do gradiente e por um filtro gaussiano. É utilizada uma função gaussiana com σ igual à metade do peso da janela do descritor para atribuir um peso à magnitude de cada ponto amostral, que está ilustrado como uma janela circular no lado esquerdo da Figura 2.2. O propósito dessa janela gaussiana é evitar mudanças bruscas no descritor com pequenas mudanças na posição da janela e dar uma ênfase menor aos gradientes que estão distantes do centro do descritor, pois esses são os mais afetados por erros.

O descritor principal é mostrado no lado direito da Figura 2.2. Para computá-lo, as

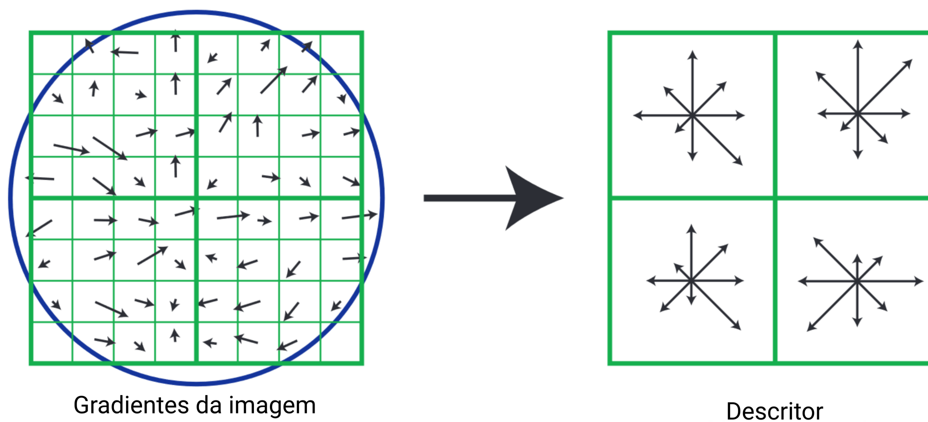

Figura 2.2: Um descritor é criado a partir da magnitude do gradiente e orientação de cada ponto amostral da imagem na região em torno do ponto de interesse, como mostrado à esquerda. Isto é ponderado por uma janela gaussiana, indicada pelo círculo. As amostras são acumuladas em histogramas de orientação sumarizando o conteúdo de regiões 4x4, como mostrado à direita, com o tamanho de cada vetor correspondendo à soma das magnitudes do gradiente próximas à direção da região. A figura ilustra um descritor 2x2 computado a partir de amostras 8x8. (Fonte: [24]).

amostras são acumuladas em histogramas de orientação que sumarizam o conteúdo em regiões 4x4 (mostrado de forma simplificada no exemplo 2x2 da figura). O gradiente de orientações é quantizado em 8 posições para cada região, o que leva a um descritor SIFT de 128 dimensões. O uso de gradientes e a discretização espacial da imagem torna esse descritor bastante robusto a distorções e ruídos, bem como a variações aditivas na imagem.

2.1.2 Representação por Meio de Descritores Locais

Existem dois tipos de representação baseados em descritores locais: (i) Representação global (agregado) e (ii) Conjunto de características locais (não agregado). Para cada um devem ser escolhidos métodos de comparação diferentes. Representações globais tipicamente utilizam a aprendizagem de um classificador para fazer a comparação entre os descritores ou métodos de agregação de descritores em um vetor global, enquanto representações locais tipicamente calculam um escore de similaridade.

Para o cálculo de escores de similaridade é necessário computar escores individuais entre os descritores locais e agregá-los em um escore global. Alguns dos métodos utilizam essa forma de representação são o Hamming Embedding (HE) [28] e o Naive-Bayes Nearest-Neighbor (NBNN) [29].

A obtenção de um descritor global pode ser feita pela codificação dos descritores locais seguida pela agregação dos mesmos em um único vetor. Isso permite a comparação direta de imagens em consultas, o que contribui para a escalabilidade, já que não há a necessidade de salvar informação relacionada à correspondência de descritores. Duas importantes representações são o Bag of Words (BOW) [30, 31] e o Vector of Locally Aggregated Descriptors (VLAD) [32], que devido à sua relevância em aplicações de recuperação de imagens serão detalhadas a seguir.

BOW

Bag of Words (BOW) é uma forma de representação de imagens por meio da agregação de descritores locais utilizando quantização de vetores. Ela foi introduzida por Sivic e Zisserman [30] e, paralelamente, utilizada por Csurka *et al.* [31] para classificação de imagens. Desde então foi amplamente utilizada no reconhecimento de imagens e aprimorado para soluções em recuperação e classificação de imagens. Nesse método, a agregação dos descritores locais em um vetor global é feita a partir de três passos [6]: (i) criação do *codebook*, (ii) codificação e (iii) agregação.

O primeiro passo é a criação do *codebook*, que consiste em um conjunto de vetores conhecidos como *visual words*. Eles são criados a partir de um grande conjunto de descritores locais utilizando clusterização não-supervisionada para particionar o espaço em K clusters. Cada cluster é associado a um vetor de representação, o conjunto desses vetores compõe o vocabulário, ou seja, são as *visual words*. Para a clusterização foram utilizados diversos métodos, desde o popular método *k-means* [30] até o Gaussian Mixture Model (GMM) [33].

A codificação dos descritores é feita a partir das *visual words*. A forma mais simples é atribuir a *visual word* mais próxima a cada descritor. Essa forma de quantização é a mais simples e provavelmente a mais popular, no entanto essa abordagem leva a erros de quantização e perde informação. Outro fator sensível desse estágio é a escolha do tamanho K do vocabulário. Se for escolhido um K muito pequeno, *visual words* diferentes podem ser atribuídos a vetores semelhantes. Já a escolha de um K muito grande pode levar à atribuição da mesma *visual word* a vetores muito diferentes.

Para limitar os erros de quantização foram propostas diversas técnicas utilizando uma atribuição suave [34], como o *Kernel Codebook* [35, 36]. Nele cada descritor é atribuído para todas as *visual words* com pesos proporcionais a $exp(-\frac{d^2}{2\sigma^2})$, onde d é a distância entre o centro do cluster e o descritor. Outra técnica de codificação é a *Super-vector coding* [37], que é similar ao VLAD, que discutiremos mais a frente.

O último estágio é a agregação das características codificadas em um vetor global. São utilizadas duas formas para isso: o *sum-pooling* e o *max-pooling*. No *sum-pooling*, as

características codificadas são combinadas em um *bin*, ou seja, em uma parte do vetor final. Por exemplo, no caso de um BOW utilizando histograma, a soma das características correspondentes a uma mesma *visual word* é adicionada ao vetor global. No *max-pooling*, o maior valor entre as características é atribuído a cada *bin*. Avila *et al.* [38] propuseram uma nova forma de fazer a agregação, chamada Bag Of Statistical Sampling Analysis (BOSSA). Nessa abordagem, os descritores são agregados por cluster baseado nas distâncias entre os centros dos clusters e os histogramas resultantes da concatenação de todos os clusters.

VLAD

VLAD é um método de representação compacta de imagens, introduzido por Jégou *et al.* [32]. Assim como em BOW, esse método aprende um *codebook* de K *visual words* utilizando o *k-means*. Cada descritor é associado à *visual word* mais próxima. A ideia do VLAD é acumular para cada *visual word* os resíduos entre os vetores atribuídos à ela e o centro do cluster. A concatenação dos resíduos gera um vetor global de dimensão $D \times K$, onde D é a dimensão dos vetores de características.

O descritor é representado por $v_{i,j}$, onde os índices $i = 1...K$ e $j = 1...D$ respectivamente, indexam a *visual word* e o componente do descritor local. Dessa forma, o componente v é obtido a partir da soma de todos os descritores:

$$v_{i,j} = \sum_{x \, tal \, que \, NN(x) = c_i} x_j - c_{i,j} \tag{2.1}$$

onde x_j e $c_{i,j}$ denotam, respectivamente, o j-ésimo componente do descritor x e sua *visual word* c_i correspondente. O vetor v é então L2-normalizado por $v := v/||v||_2$.

Devido a sua simplicidade e eficiência, o VLAD se tornou uma escolha popular em aplicações de recuperação de imagens. Em [39] Jégou *et al.* introduz uma normalização para o VLAD. Apesar de ter sido proposto inicialmente para recuperação de imagens, essa representação também pode ser utilizada para classificação, como em [40].

2.1.3 Descritores Globais

A extração de descritores globais para representação de imagens é uma escolha menos popular, por tipicamente comprometer a busca por similaridade com informações ruidosas incluídas no descritor. Em busca de driblar esses problemas e manter a escalabilidade da representação global, métodos como BOW e VLAD fazem a agregação de descritores locais em um mesmo vetor.

No entanto, algumas abordagens modernas de extração de descritores globais têm obtidos bons resultados em comparação com descritores locais agregados. Uma delas é o GIST [41], um descritor global de baixa dimensionalidade, que não exige nenhuma forma de segmentação e, quando comparado ao BOW, obteve resultados satisfatórios em buscas [42]. Outra forma de representação é o Fuzzy Color and Texture Histogram (FCTH) [43], nele são combinadas informações referentes a cor e textura em um mesmo histograma. Essa abordagem mostrou bons resultados quando comparado ao GIST [44].

GIST

Pesquisas da psicologia mostram que o ser humano consegue captar a essência (*gist*) de uma imagem apenas a vendo de relance por alguns segundos. A extração do descritor GIST segue esse princípio, o método proposto por Siagian e Itti [45] tem dois passos principais: construção de mapa de característica de saliência e extração do descritor GIST.

O primeiro passo consiste na construção de mapas baseados em características visuais de baixo nível, como as cores, a intensidade, e a orientação do canal. No total são computados 34 mapas de característica de saliência: 6 de intensidade, 12 de cores e 16 de orientação.

Então cada mapa é dividido em subregiões 4x4, e a média de cada subregião é utilizada para a construção dos descritores GIST de dimensão 16. A combinação dos vetores de cada mapa gera um vetor global de dimensão 544. Dessa forma cada imagem pode ser representada por esse descritor.

2.2 Recuperação de Imagens

2.2.1 Busca por Similaridade

Inicialmente a consulta em bases de dados multímida era feita principalmente por meio de elementos textuais, como palavras-chaves ou títulos. Com aumento no número de aplicações que utilizam bases de dados multimídia e no volume dos dados, surgiu a busca por similaridade baseada em conteúdo [46]. Esse método propõe uma forma de encontrar objetos da base de dados que são similares a um dado objeto de consulta.

Esse tipo de consulta, baseada em conteúdo, utiliza a extração de características para eliminar a distância entre o código de baixo nível e as complexas relações semânticas dos dados. A ideia básica é extrair características importantes do objeto, mapeá-las em vetores de alta dimensionalidade e realizar a busca de objetos com vetores de características similares na base de dados (Figura 2.3).

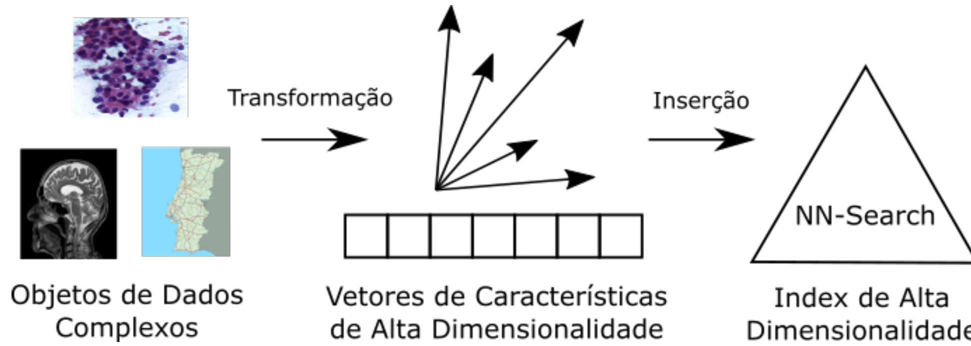

Figura 2.3: Ideia básica da busca por similaridade baseada em características.

A transformação de características consiste no mapeamento do objeto multimídia em um vetor de características de dimensão d

$$F : Obj \rightarrow R^d \tag{2.2}$$

e a similaridade entre dois objetos é dada pela distância Euclidiana entre os vetores de características dos objetos.

$$\delta(obj_1, obj2) = \delta_{Euclidiana}(F(obj_1), F(obj2)) \tag{2.3}$$

A busca consiste em calcular a distância entre o vetor de consulta e todos os vetores da base, ranquear as distâncias e retornar os mais próximos (NN), ou seja, que são mais similares. Os principais problemas dessa abordagem são a alta dimensionalidade dos vetores e as grandes bases empregadas, o que compromete em muito o desempenho do algoritmo.

2.2.2 Indexação e Busca Aproximada

É muito comum a adoção de duas abordagens para minimizar o custo da busca por similaridade: (i) uso de algoritmos de indexação e estruturas de dados que reduzam o espaço de busca, particionando os objetos espacialmente e (ii) busca aproximada de vizinhos mais próximos (ANN), que substitui o cálculo exato dos vizinhos 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 [11, 47, 48], árvores *k-means* [49, 12], árvores de cobertura [50], dentre outros.

O trabalho de Friedman *et al.* introduz uma forma de utilizar árvores k-d em buscas por vizinhos mais próximos [11]. A árvore k-d é binária, em que cada nó representa uma

partição de um arquivo. 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 a 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 [47] e propõem uma estratégia de busca dos melhores conjuntos da árvore binária, a Best Bin First (BBF).

Outro método é o uso de árvores *k-means*, proposto por Fukunaga e Narendra [49]. 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 [12] que, de forma semelhante ao proposto em [47], 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 Fast Library for Approximate Neighbors Search (FLANN) [12, 51, 19, 52], Locality-Sensitive Hashing (LSH) [15], *Multicurves* [53] e Product Quantization for Approximate Nearest Neighbor Search (PQANNS) [16].

FLANN

FLANN é um sistema proposto por Muja e Lowe que seleciona automaticamente o melhor entre diferentes algoritmos de busca. Os algoritmos implementados no sistema utilizam uma das seguintes abordagens: árvores randômicas k-d ou árvore *k-means* com busca por prioridade.

O algoritmo de árvores randômicas k-d se baseia no trabalho de Silpa-Anan e Hartley [48], que propõe o uso de árvores k-d criadas randomicamente para fazer a indexação e busca aproximada dos dados. No algoritmo clássico de árvore k-d [11], os dados são divididos ao meio em cada nível da árvore na dimensão para qual os dados exibem maior variação. Já as árvores k-d randômicas são construídas escolhendo a dimensão aleatoriamente entre as N_D dimensões com maior variação (Algoritmo 1).

Algoritmo 1: CONSTRUINDO AS ÁRVORES RANDÔMICAS K-D

```
   Entrada: base de dados D, número de árvores N
   Saída: a estrutura de dados de árvores randômicas k-d
 1 Procedimento ConstruirArvoresRandomicasKD(D,N):
 2 |   arvores ← {}
 3 |   para i ← {1..N} faça
 4 |   |   arvores ← ConstruirArvoreKD(D)
 5 |   fim
 6 |   retorna arvores
 7 Procedimento ConstruirArvoreKD(D):
 8 |   se |D| == 1 então
 9 |   |   CriarNoFolha(D)
10 |   senão
11 |   |   mediasEixos[1..d] ← media das características ao longo de cada dimensão
12 |   |   varianciasEixos[1..d] ← variância das características ao longo de cada
   |   |     dimensão
13 |   |   varianciaMaxEixos[1..N_d] ← N_d dimensões com maior variância
14 |   |   divisorDimensao ← uma dimensão aleatória de varianciaMaxEixos
15 |   |   divisorValor ← mediasEixos[divisorDimensao]
16 |   |   (D_1, D_2) ← divide a base de dados na dimensão divisorDimensao usando
   |   |     o valor divisorValor
17 |   |   T_1 ← ConstruirArvoreKD(D_1)
18 |   |   T_2 ← ConstruirArvoreKD(D_2)
19 |   |   CriarNoInterno(divisorDimensao, divisorValor, T_1, T_2)
20 |   fim
```

A busca em árvores randômicas k-d é realizada a partir de uma única fila de prioridade partilhada entre as árvores. Essa fila de prioridade contém todos os ramos não explorados ordenados de forma crescente pela distância entre sua região correspondente e o vetor de consulta. Dessa forma a busca explora primeiro as folhas mais próximas da consulta

em todas as árvores. Uma vez que um ponto é examinado, ele é marcado para não ser examinado em outra árvore. O grau de aproximação é determinado pelo número máximo de folhas a serem visitadas, retornando os melhores candidatos à vizinhos mais próximos encontrados até aquele momento (Algoritmo 2).

Algoritmo 2: BUSCANDO EM ÁRVORES RANDÔMICAS K-D

```
   Entrada: árvores randômicas k-d T_i, vetor de consulta Q, quantidade de
            vizinhos K, quantidade máxima de pontos a serem examinados L
   Saída: K vizinhos mais próximos aproximados
 1 Procedimento BuscarEmArvoresKD(T_i,Q, K, L):
 2 |  contador ← 0
 3 |  PQ ← fila de prioridade vazia
 4 |  R ← fila de prioridade vazia
 5 |  para cada árvore T_i faça
 6 |  |  T ← topo de PQ
 7 |  |  AtravessarArvoresKD(Q, T, PQ, R)
 8 |  fim
 9 |  enquanto PQ não estiver vazio e contador < L faça
10 |  |  T ← topo de PQ
11 |  |  AtravessarArvoresKD(Q, T, PQ, R)
12 |  fim
13 |  retorna K pontos superiores de R
14 Procedimento AtravessarArvoresKD(Q, T, PQ, R):
15 |  se T é um nó folha então
16 |  |  adicione ponto em R
17 |  |  contador ← contador + 1
18 |  senão
19 |  |  divisorDimensao ← T.divisorDimensao
20 |  |  divisorValor ← t.divisorValor
21 |  |  se Q(divisorDimensao) < divisorDimensao então
22 |  |  |  noMaisProximo ← T.esquerda
23 |  |  |  outroNo ← T.direita
24 |  |  senão
25 |  |  |  noMaisProximo ← T.direita
26 |  |  |  outroNo ← T.esquerda
27 |  |  fim
28 |  |  adicione outroNo a PQ
29 |  |  AtravessarArvoresKD(Q, noMaisProximo, PQ, R)
30 |  fim
```

Apesar das árvores randômicas k-d serem muito eficazes em muitas situações, em algumas bases de dados a árvore *k-means* com busca por prioridade é mais eficiente, especialmente quando é necessária uma precisão alta.

A árvore *k-means* com busca por prioridade tenta explorar melhor a estrutura natural dos dados, através da clusterização dos pontos usando a distância completa entre todas as dimensões. Diferentemente das árvores randômicas k-d, que particionam uma dimensão por vez.

A construção da árvore *k-means* com busca por prioridade é feita através do particionamento dos dados em k regiões distintas utilizando o algoritmo de clusterização *k-means*, e então o aplicando recursivamente aos pontos de cada região. O particionamento para quando o número de pontos em uma região é menor que k (Algoritmo 3).

Algoritmo 3: CONSTRUINDO A ÁRVORE K-MEANS COM BUSCA POR PRIORIDADE

Entrada: base de dados D, fator de ramificação K, quantidade máxima de iterações I_{max}, algoritmo de seleção de centro a utilizar C_{alg}

Saída: árvore *k-means*

```
1  se |D| < K então
2  │ criar nós folha com os pontos de D
3  senão
4  │ P ← selecione os K pontos de D usando o algoritmo C_alg
5  │ convergiu ← falso
6  │ iteracoes ← 0
7  │ enquanto não convergiu e iteracoes < I_max faça
8  │ │ C ← agrupe os pontos em D em torno dos centros mais próximos P
9  │ │ P_novo ← médias dos clusters em C
10 │ │ se P = P_novo então
11 │ │ │ convergiu ← verdadeiro
12 │ │ fim
13 │ │ P ← P_novo
14 │ │ iteracoes ← iteracoes + 1
15 │ fim
16 │ para cada cluster C_i ∈ C faça
17 │ │ criar uma nó interno com centro em P_i
18 │ │ aplicar recursivamente o algoritmo nos pontos em C_i
19 │ fim
20 fim
```

Para a busca, a árvore é percorrida da raiz até o nó mais próximo da consulta, seguindo cada nó interno com o centro do cluster mais próximo da consulta, e adicionando os nós não explorados na fila de prioridade PQ. A fila de prioridade é ordenada pela distância entre a consulta e o limite do ramo que está sendo adicionado na fila. Em seguida o algoritmo segue percorrendo a árvore, agora iniciando com o nó do topo da fila de prioridade (Algoritmo 4). Isso é feito até que uma quantidade predeterminada de folhas sejam visitadas, de forma semelhante à busca em árvores randômicas k-d.

Algoritmo 4: BUSCANDO NA ÁRVORE K-MEANS COM BUSCA POR PRIORIDADE

Entrada: árvore *k-means* T, vetor de consulta Q, quantidade máxima de pontos a serem examinados L

Saída: K vizinhos mais próximos aproximados

```
1  Procedimento BuscarEmArvoreKMeans(T,Q, L):
2  |  contador ← 0
3  |  PQ ← fila de prioridade vazia
4  |  R ← fila de prioridade vazia
5  |  AtravessarArvoreKMeans(T, PQ, R) enquanto PQ não estiver vazio e
   |    contador < L faça
6  |  |  N ← topo de PQ
7  |  |  AtravessarArvoreKMeans(N, PQ, R)
8  |  fim
9  |  retorna K pontos superiores de R
10 Procedimento AtravessarArvoreKMeans(N, PQ, R):
11 |  se N é um nó folha então
12 |  |  busque todos os pontos em N e os adicione em R
13 |  |  contador ← contador + |N|
14 |  senão
15 |  |  C ← nós filhos de N
16 |  |  C_q ← nó de C mais póximo da consulta Q
17 |  |  C_p ← C C_q
18 |  |  adicione todos os nós em C_p a PQ
19 |  |  AtravessarArvoreKMeans(C_q, PQ, R)
20 |  fim
```

Cada um desses algoritmos tem suas vantagens, dependendo do tipo de base de dados com qual trabalha. Diversos fatores vão influenciar a escolha do algoritmo mais apropriado, tais como a dimensionalidade dos dados, tamanho e estrutura da base de dados e a precisão de busca desejada [52]. Além disso, cada algoritmo tem um conjunto de parâmetros que influenciam significantemente o desempenho da busca. Esses parâmetros incluem o número de árvores randômicas a serem utilizadas, no caso de árvores kd, ou o fator de ramificação e o número de iterações no caso da árvore *k-means* com busca por prioridade.

Geralmente a escolha de parâmetros é feita manualmente, por meio de diferentes heurísticas e raramente de uma forma sistemática. O FLANN implementa a seleção automática do melhor algoritmo de busca de vizinhos mais próximos para uma base de dados específica, bem como seus parâmetros ótimos.

Considerando o algoritmo de busca de vizinhos mais próximos como um parâmetro de uma rotina genérica de busca A, o problema se reduz à determinação dos parâmetros

$\theta \in \Theta$ que dão a melhor solução, onde θ é conhecido como o espaço de configuração de parâmetros. Isso pode ser formulado como um problema de otimização do espaço de configuração de parâmetros:

$$\min_{\theta \in \Theta} c(\theta), \quad (2.4)$$

em que $c : \Theta \leftarrow \mathbb{R}$ é a função de custo indicando o quão bem o algoritmo de busca A, configurado com os parâmetros θ, desempenha na base de entrada.

O custo é definido pela combinação de três fatores: o tempo de busca, o tempo de construção da árvore e a sobrecarga de memória. Dependendo da aplicação, cada um desses fatores pode ter um nível de importância: quando a árvore é construída apenas uma vez para uma grande quantidade de consultas, o tempo de construção da árvore não é tão relevante. Se a árvore é construída on-line e consultada poucas vezes, tanto o tempo de construção da ávore quanto o tempo de busca são importantes. E há casos onde é interessante limitar a sobrecarga de memória, dependendo da disponibilidade de recursos para a execução do algoritmo. A função de custo é definida a seguir:

$$c(\theta) = \frac{s(\theta) + w_b b(\theta)}{\min_{\theta \in \Theta}(s(\theta) + w_b b(\theta))} + w_m m(\theta), \quad (2.5)$$

onde $s(\theta)$, $b(\theta)$ e $m(\theta)$ representam o tempo de busca, tempo de construção da árvore e a sobrecarga de memória utilizando os parâmetros θ, respectivamente. A sobrecarga de memória é calculada pela razão entre a memória usada pela(s) árvore(s) e e a memória usada pelos dados: $m(\theta) = m_t(\theta)/m_d$.

Os pesos w_b e w_m são usados para controlar a importância relativa da construção de árvores e da sobrecarga de memória. O peso w_b controla a importância do tempo de construção da árvore em relação ao tempo de busca. O tempo de busca é definido como o tempo gasto para realizar um número de consultas igual à quantidade de pontos contidos na árvore.

A otimização proposta é realizada em dois passos: uma exploração global do espaço de parâmetros utilizando *grid search*, seguido de uma otimização local começando com a melhor solução encontrada no primeiro passo. A *grid search* é possível e efetiva porque a quantidade de parâmetros é relativamente pequena. No segundo passo é utilizado o método de Nelder-Mead [54] para explorar localmente o espaço de parâmetros e aprimorar a melhor solução obtida no primeiro passo.

No FLANN, a otimização pode ser executada em todos os dados para obter resultados mais acurados ou apenas em uma fração dos dados para fazer a configuração de parâmetros mais rápido. A seleção de parâmetros precisa ser executada apenas uma vez para cada tipo de base de dados e os valores ótimos podem ser utilizados em execuções posteriores.

LSH

O LSH é uma técnica proposta por Indyk e Motwani [55] que utiliza funções para resolver o problema da "maldição da dimensionalidade". Dada uma base de dados contendo os vetores $x_1, ..., x_n$ e um vetor de consulta q, a ideia básica do LSH é projetar os dados em um espaço binário de baixa dimensionalidade (espaço de Hamming), isto é, mapear cada ponto para um vetor de b bits, chamado chave *hash*. Se a projeção for realizada corretamente, é possivel encontrar vizinhos mais próximos aproximados em tempo sublinear a n. As chaves *hash* são construídas aplicando b funções de *hash* $h_1, ..., h_b$ para cada objeto da base. Para ser válida, cada função deve satisfazer a seguinte propriedade:

$$Pr[h(x_i) = h(x_j)] = sim(x_i, x_j), \tag{2.6}$$

onde $sim(x_i, x_j) \in [0, 1]$ é a função de similaridade de interesse. Isso garante que a função agrupa pontos semelhantes no mesmo balde e permite que a consulta seja feita apenas nos vetores contidos no balde ao qual o vetor de consulta foi assinalado.

Para garantir uma boa qualidade dos resultados, é comum que sejam combinadas uma quantidade de funções de *hash* $h \in \mathcal{H}$ em uma função de *hash* g. Além disso, é definida uma família $\mathcal{G}$ de funções $g(v) = (h_1(v), ..., h_k(v))$, onde $h_i \in \mathcal{H}$. O algoritmo então escolhe aleatóriamente L funções $g_1, ..., g_L$ de $\mathcal{G}$ e cria L vetores de *hash*, um para cada função. No pré-processamento, cada ponto da base de dados é armazenado em $g_j(v)$ baldes, para todo $j = 1, ..., L$.

Para realizar uma consulta q, o algoritmo calcula $g_1(q), ..., g_L(q)$ e busca nos baldes assinalados pelas funções de *hash*. Para cada ponto armazenado nos baldes, é calculada a distância entre eles e a consulta q e são retornados os mais próximos.

Multicurves

Curvas de preenchimento de espaço são curvas que tem a capacidade de preencher um espaço multidimencional, o que permite o mapeamento de vetores d-dimensionais em pontos unidimensionais da curva. Essa propriedade foi primeiramente utilizada de forma explícita na busca aproximada de vizinhos mais próximos por Faloutsos [56], utilizando curvas *Gray-code*. De forma independente Skubalska-Rafajłowicz e Krzyżak fizeram uso da curva de Sierpiński para realizar buscas de vizinhos mais póximos [57].

A ideia desses métodos é mapear os vetores multidimensionais da base de dados em pontos da curva, obtendo uma coordenada unidimensional chamada chave extendida, que representa sua posição relativa no tamanho da curva. A chave extendida é usada para realizar a busca por similaridade unidimensional. No entanto, isso considera que pontos próximos na curva estão necessariamente próximos na curva, o que não é verdade. Nem

todos os pontos que estão próximos no espaço, estão mapeados em trechos próximos da curva. Isso se dá por efeitos de borda, que tendem a colocar pontos próximos no espaço em regiões muito distantes da curva Figura 2.4.

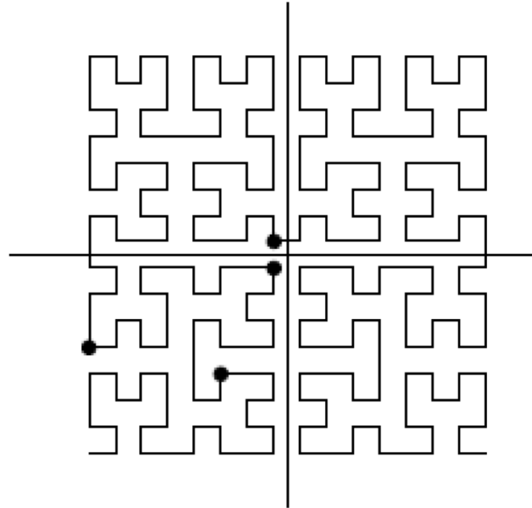

Figura 2.4: O problema dos efeitos de borda em curvas de preenchimento de espaço. Os pontos no centro da imagem estão bem mais distantes na curva que os pontos do quadrante inferior à esquerda, mesmo que estejam mais próximos no espaço. (Fonte: [53]).

Para contornar esse problema, Valle *et al.* propõem o uso de múltiplas curvas (*Multicurves*) para realizar a aproximação dos vetores [53, 58, 59]. Esse método consiste na aproximação de subconjuntos de dimensões dos vetores por meio de curvas, isto é, são executadas diversas projeções dos subconjuntos das dimensões do dado para um espaço unidimensional a partir das curvas. Na busca, é feita a decomposição do vetor de consulta e computadas as chaves extendidas de cada subdimensão. Então, para cada subdimensão são explorados os elementos contendo as chaves extendidas mais próximas à da chave extendida da consulta.

PQANNS

Um método muito bem sucedido é o PQANNS [16], que é utilizado neste trabalho e apresentado com mais detalhes no Capítulo 3. Ele consiste na quantização dos vetores de características e na clusterização dos dados da base para indexação a partir de uma estrutura de lista invertida. Na quantização, o espaço vetorial é decomposto em subespaços de menor dimensionalidade que são quantizados separadamente. O vetor passa então a ser representado por um código reduzido, composto pelos índices de quantização de cada subespaço.

A indexação é feita a partir da criação de clusters por meio de uma base de treinamento, os índices dos centroides de cada cluster são associados às entradas de uma lista invertida. É calculada a distância euclidiana entre cada vetor da base e os centroides dos clusters, então cada vetor é associado ao centroide mais próximo e seu índice e vetor quantizado são incluídos na entrada correspondente da lista invertida. Para a realização da busca, é calculada a distância entre o vetor quantizado de consulta e os centroides da lista invertida. Os vetores das w entradas mais próximas são buscados e os mais próximos são retornados na busca.

O PQANNS mostra resultados superiores aos de outros algoritmos, especialmente em relação ao uso de memória. Em [16], Jégou *et al.* mostra que para uma mesma qualidade de busca, o PQANNS realiza a busca mais rapidamente que o FLANN e sua estrutura de indexação usa cerca de 10 vezes menos memória.

2.3 Arquiteturas Paralelas

As arquiteturas paralelas tem como propósito prover uma estrutura de alto nível para o desenvolvimento de soluções paralelas utilizando múltiplas unidades de processamento, que cooperam na realização de problemas concorrentemente [60]. Para isso o problema é decomposto em partes menores que podem ser resolvidas simultaneamente por unidades de processamento diferentes.

Dentre as diversas arquiteturas propostas é possível encontrar alguns padrões, isso levou à criação de taxonomias que classifiquem as arquiteturas. As mais relevantes são as propostas por Flynn [61] e por Duncan [60]. Nas Subseções 2.3.1 e 2.3.2 essas arquiteturas serão explicadas em mais detalhes.

2.3.1 Taxonomia de Flynn

A taxonomia de Flynn classifica arquiteturas a partir da presença de um ou múltiplos fluxos de dados e instruções. Isso leva às quatro categorias abaixo:

- Single Instruction, Single Data Stream (SISD) - Um único fluxo de instruções para um fluxo de dados. Compreende a arquitetura de máquinas seriais, não paralelas.
- Multiple Instruction, Single Data Stream (MISD) - Múltiplos fluxos de instruções para um único fluxo de dados.
- Single Instruction, Multiple Data Streams (SIMD) - Um único fluxo de instruções para múltiplos fluxos de dados. Essa arquitetura explora o paralelismo a nível de dados.

- Multiple Instruction, Multiple Data Streams (MIMD) - Múltiplos fluxos de instruções para múltiplos fluxos de dados. Pode-se incluir nessa categoria os clusters e multiprocessadores com memória compartilhada.

2.3.2 Taxonomia de Duncan

Duncan propõe uma taxonomia mais detalhada e hierárquica, em busca de sanar alguns problemas de classificação de arquituras mais modernas e complexas. Essa classificação se baseia nas principais arquiteturas paralelas e pode ser visualizada na Figura 2.5.

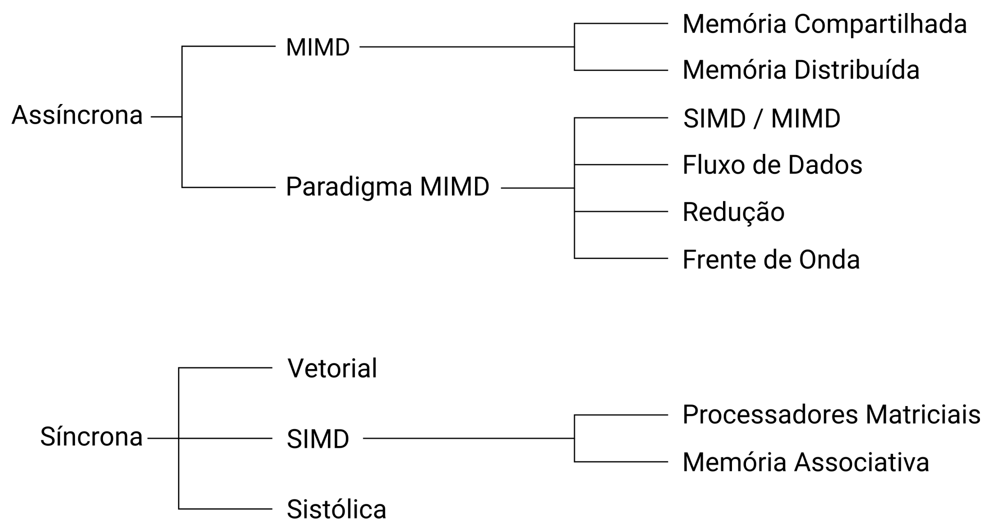

Figura 2.5: Taxonomia de Duncan (Fonte: [60]).

- Arquiteturas Síncronas - são coordenadas por meio de operações concorrentes que executam as mesmas instruções em paralelo por meio de clocks globais, unidades centrais de controle ou controladores de unidade vetorial.
  - Vetorial: executam operações em vetores de dados.
  - SIMD: tipicamente emprega uma unidade central de controle, múltiplos processadores e uma interconexão processador-processador ou processador-memória. A unidade de controle central envia uma instrução para todos os processadores, que executam a instrução nos dados locais. Essa arquitetura pode ser dividida em outras duas subcategorias: Arranjo de Processadores e Memória Associativa.
    * Arranjo de Processadores - essa arquitetura implementa instruções que operam diretamente com vetores, ao invés de fazer o processamento escalar.

    * Memória Associativa - Nessa arquitetura o acesso a memória é vinculado ao seu conteúdo, ao invés do endereço.

  - Sistólica: busca suprir necessidades de computação intensiva que podem possuir operações de Entrada e Saída (E/S). Para isso são utilizados processadores conectados linearmente, que executam um conjunto de operações a cada período.

- MIMD - emprega múltiplos processadores que podem executar diferentes fluxos de instruções de forma independente, usando dados locais. É dividida em duas subcategorias: Memória Distribuída e Memória Compartilhada.

  - Memória Distribuída - é composta por unidades de processamento autônomas com sua própria memória local, que se comunicam por meio da troca de mensagens na rede.
  - Memória Compartilhada - a memória é global e compartilhada entre os processadores, que podem fazer acesso aos dados concorrentemente.

- Paradigma MIMD - São arquiteturas com características específicas além das que caracterizam as arquiteturas MIMD.

  - MIMD/SIMD - Possuem trechos da arquitura MIMD que funcionam sob a arquitetura SIMD.
  - Fluxo de Dados - Essa arquitetura segue um paradigma de execução onde as instruções são habilitadas para execução assim que todos os operandos estão disponíveis. Assim, a sequência de instruções a serem executadas dependem dos dados, o que permite a exploração de concorrência em nível de tarefas, rotinas e instruções.
  - Redução - Nessa arquitetura as instruções são habilitadas para execução assim que seus resultados são requisitados por outra instrução já habilitada para execução.
  - Frente de Onda - Combina vários processadores linearmente com um fluxo de dados assíncrono, de forma em que o acesso aos dados seja feito de acordo com a disponibilidade dos operandos de cada operação.

2.4 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 [19, 17, 18, 20, 62, 63, 21, 64, 65].

Dentre os trabalhos propostos, pode-se destacar as paralelizações do LSH utilizando o MapReduce [17, 18] e a versão paralela do FLANN em cima do MPI [19].

2.4.1 Paralelizações do LSH em MapReduce

Stupar *et al.* [17] usam uma estratégia de MapReduce em sua paralelização do LSH. No estágio de mapeamento são visitados todos os baldes a que o vetor de consulta foi assinalado no *hash*, gerando um conjunto de candidatos a vizinhos mais próximos de cada balde. Na redução os candidatos dos baldes são agregados e o resultado global é calculado. Essa implementação do LSH armazena os baldes em um sistema distribuído de arquivo (HDFS) utilizando um arquivo por balde.

No entanto as combinações de parâmetros do LSH podem criar uma grande quantidade de arquivos (baldes), o que reduz o desempenho do sistema. Além disso, é armazenado o conteúdo dos objetos em cada arquivo (balde), ao invés do identificador. Isso faz com que a base de dados seja replicada para cada tabela de *hash*. Uma implementação eficiente do LSH pode gerar centenas de tabelas. Esse nível de replicação é proibitivo para grandes bases de dados.

Bahmani *et al.* [18] 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) ativo para o armazenamento em memória. Eles propõem 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 [15]. 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.

2.4.2 Paralelização do FLANN em MPI

Muja *et al.* propõem uma versão paralela do FLANN em cima do MPI [19]. 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.

Capítulo 3

Product Quantization for Approximate Nearest Neighbor Search (PQANNS)

A abordagem Product Quantization for Approximate Nearest Neighbor Search (PQANNS) contorna o problema da "maldição da dimensionalidade", propondo a decomposição do espaço em um produto cartesiano de subespaços de baixa dimensionalidade para quantização futura. Além disso, para evitar a busca exaustiva é utilizada uma lista invertida como estrutura de indexação.

O sucesso dessa abordagem se deve à garantia de um compromisso entre a qualidade da busca e eficiência, enquanto faz uso de uma estrutura de indexação de baixo uso de memória. Na Seção 3.1 vamos detalhar a decomposição do espaço e a quantização dos subespaços. Na Seção 3.2 são apresentadas formas de cálculo de distâncias em espaços quantizados, bem como a utilizada em PQANNS. Em seguida, na Seção 3.3 são abordados os conceitos relacionados à busca não exaustiva em espaços quantizados (PQANNS).

3.1 Quantização de Espaços

O cálculo de distâncias euclidianas entre vetores de alta dimensão é fundamental na busca por vizinhos mais próximos (Nearest Neighbor (NN)), pois é a principal métrica utilizada para definir a semelhança entre os vetores. No entanto, esse processo é encarecido com o aumento da dimensionalidade dos vetores. Para contorná-lo é possível fazer um cálculo aproximado das distâncias a partir da quantização dos vetores.

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. 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. Dessa forma, cada vetor do conjunto V é reconstruído a partir do centroide que representa a célula a que ele pertence. A qualidade da quantização pode ser medida pelo cálculo do Erro Quadrático Médio (EQM) entre o vetor de entrada x e seu valor quantizado $q(x)$.

Para a quantização ser ótima, ela deve obedecer duas propriedades conhecidas como condições de otimalidade de Lloyd. A primeira condição diz que o vetor x deve ser quantizado para os centroides mais próximos, em termos de distância Euclidiana:

$$q(x) \ = \ arg\ min\ d(x, c_i),\ \ c_i \in C. \tag{3.1}$$

Com isso, as células devem ser delimitadas por hiperplanos. A segunda condição de Lloyd diz que o valor reconstruído deve ser o esperado por vetores situados na célula de Voronoi.

O quantizador de Lloyd, que corresponde ao algoritmo de clusterização *k-means*, encontra um *codebook* próximo do ótimo por meio do particionamento iterativo de uma base de treinamento. Ele encontra os centroides de cada conjunto de acordo com a proximidade dos vetores. Como a quantização é feita utilizando o *kmeans*, as condições de Lloyd são obedecidas e é garantido que será encontrado um ótimo local em termos de erro de quantização.

Vamos considerar um vetor de 128 dimensões, como o descritor SIFT. Um quantizador produzindo códigos de 64 bits, isto é, 0.5 bit por componente, 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 (3.1). Cada subvetor u_j, $1 \leq j \leq m$ com $D^* = D/m$ dimensões. A quantização de x pode ser representada por:

$$\underbrace{x_1, ..., x_{D^*}}_{u_1(x)}, ..., \underbrace{x_{D-D^*+1}, ..., x_D}_{u_m(x)} \rightarrow q_1(u_1(x)), ..., q_m(u_m(x)) \tag{3.2}$$

Por fim, a quantização de x será o produto cartesiano de seus subvetores quantizados:

$$q(y) = q_1(u_1) \times q_2(u_2) \times ... \times q_m(u_m) \tag{3.3}$$

Essa abordagem permite que a quantização de baixa complexidade de cada subvetor seja combinada para criar uma indexação de alta complexidade, que vai endereçar um *codebook*. O *codebook* é então construído pelo produto cartesiano de múltiplos conjuntos

menores de centroides, $C = C_1 \times ... \times C_m$. Assumindo que todos os subquantizadores tem o mesmo número finito k^* de centroides, o número total de centroides será $(k^*)^m$.

Isso permite a criação de grandes *codebooks* a partir de vários conjuntos pequenos de centroides. Ao aprender os centroides com o *k-means*, apenas uma amostra do conjunto total de vetores é usada, mas ele continua sendo adaptado a distribuição dos dados que ele vai representar. A complexidade da aprendizagem dos quantizadores é m vezes a complexidade de executar a clusterização com k^* centroides de dimensão D^*.

Guardar todos os kD valores em ponto flutuante do *codebook* na memória não é eficiente, portanto o ideal é que sejam guardados os mk^* centroides de todos os subquantizadores, isto é, reduzir o consumo de memória para $mD^*k^* = k^*D$ valores. Quantizar um elemento apenas com o *k-means* requer kD operações de ponto flutuante, com o uso da quantização do produto a complexidade reduz para a ordem de k^*D.

3.2 Cálculo de Distâncias em Espaços Quantizados

Após computar a quantização, os vetores da base de dados são representados por seus índices no *codebook*. Assim, a busca pelos vizinhos mais próximos vai ser executada no espaço quantizado usando os índices do *codebook*. Para isso é necessário calcular a distância entre o vetor de consulta e os vetores da base de dados. Os autores de [16] propõem dois métodos para aproximar a distância entre o vetor de conulta x e os valores quantizados da base $(q(y))$, um simétrico e um assimétrico. Veja a Figura 3.1 para uma representação visual.

No cálculo de distância simétrica (SDC), ambos os vetores x e y são representados por seus respectivos centroides $q(x)$ e $q(y)$. A distância aproximada é então obtida por meio da Equação 3.4.

$$\hat{d}(x, y) = d(q(x), q(y)) = \sqrt{\sum_j d(q_j(x), q_j(y))^2} \tag{3.4}$$

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 é computada pela Equação 3.5.

$$\tilde{d}(x, y) = d(q(x), q(y)) = \sqrt{\sum_j d(u_j(x), q_j(u_j(y)))^2} \tag{3.5}$$

ADC busca melhorar a qualidade da aproximação ao usar x ao invés de $q(x)$ para calcular as distâncias. A complexidade da execução de ambos os métodos é equivalente, enquanto ADC apresenta valores mais fiéis à distância real. A única vantagem do SDC é

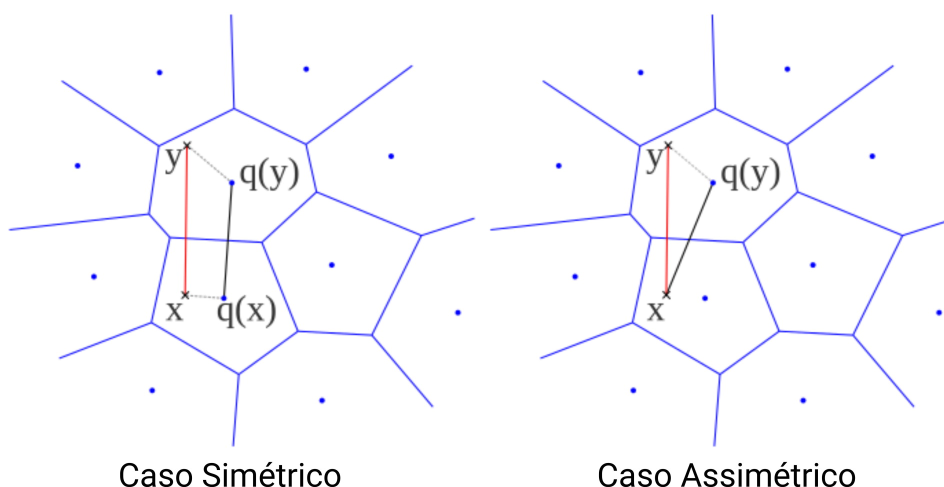

Figura 3.1: Ilustração do cálculo de distância simétrico (esquerda) e assimétrico (direita). No método simétrico, o cálculo utiliza apenas valores quantizados para determinar a distância $d(q(x), q(y))$, enquanto o assimétrico utiliza o vetor de consulta x e os valores quantizados da base de dados $d(x, q(y))$. Observe que a reta de cor vermelha corresponde à distância real, enquanto a de cor preta corresponde à distância aproximada. (Fonte: [16]).

reduzir o uso de memória dos vetores de consulta, no entanto isso é irrelevante na maior parte dos casos e o método assimétrico deve ser escolhido.

3.3 Busca Não Exaustiva em Espaços Quantizados

A busca aproximada de vizinhos mais próximos em espaços quantizados é rápida e reduz significantemente o requisito de memória para armazenar os descritores. No entanto, mesmo com espaços quantizados, usar métodos que comparam exaustivamente um vetor de consulta com todos os elementos da base de dados é caro.

Para evitar a busca exautiva, Jégou *et al.* propõem um método que combina um sistema de lista invertida com o cálculo assimétrico de distância, Inverted File System with Asymmetric Distance Computation (IVFADC). Uma lista invertida quantiza os descritores e armazena os índices das imagens em suas respectivas listas, o que permite o acesso rápido a uma pequena fração de índices de imagens. Além do índice, também é armazenado um pequeno código para cada vetor, o que acelera a busca pelo preço de alguns bits/bytes a mais por descritor. Para aumentar a acurácia da busca, o código armazenado na lista invertida é a diferença entre o vetor e seu centroide, ou seja, o resíduo de y.

A seguir o método IVFADC é detalhado, na Subseção 3.3.1 é explicada a criação de um segundo *codebook*, contendo os *coarse centroids*, que será utilizado na implementação

da lista invertida. Na Subseção 3.3.2 é explicada a indexação a partir da estrutura de lista invertida e na Subseção 3.3.3 é explicada a busca no PQANNS/IVFADC.

3.3.1 *Coarse Quantizer*

Um *codebook* é aprendido usando *k-means* em uma base de treino, produzindo o quantizador q_c, referenciado a seguir como *coarse quantizer*. Esse *codebook* será usado no cálculo do resíduo de y e na implementação da lista invertida.

No PQANNS, a quantização do produto é utilizada para obter uma representação enxuta dos descritores da base. No entanto, para relacioná-la aos *coarse centroids* a quantização é aplicada ao resíduo de y, ou seja, à diferença entre o vetor y e o *coarse centroid* que o representa.

$$r(y) = y - q_c(y) \tag{3.6}$$

O vetor é então aproximado pela soma da quantização do resíduo de y e seu *coarse centroid*

$$y_{aprox} \triangleq q_c(y) + q_p(y - q_c(y)). \tag{3.7}$$

A adição dos *coarse centroids* torna o código mais preciso, pois a quantização dos resíduos gera valores muito pequenos. Dessa forma, pode-se dizer que o *coarse quantizer* provê os bits mais significantes do código, enquanto o product quantizer corresponde aos bits menos significantes.

A distância aproximada do vetor de consulta x e o vetor da base y é computado pela distância $\ddot{d}(x, y)$ entre x e $\ddot{y}$:

$$d_{aprox}(x, y) = d(x, y_{aprox}) = d(x - q_c(y), q_p(y - q_c(y))). \tag{3.8}$$

Com q_{p_j} sendo o j-ésimo subquantizador, a seguinte decomposição estima a distância de forma eficiente:

$$d_{aprox}(x, y)^2 = \sum_j d(u_j(x - q_c(y)), q_{p_j}(u_j(y - q_c(y)))). \tag{3.9}$$

De forma análoga ao ADC, para cada subquantizador q_{p_j} as distâncias entre o vetor residual parcial $u_j(x - q_c(y))$ e todos os centroides $c_{j,i}$ de q_{p_j} são computadas e armazenadas.

3.3.2 Lista Invertida

A lista invertida é implementada a partir do *coarse quantizer*, sua estrutura é composta por um array de listas $\mathcal{L}_1$ ... $\mathcal{L}_{k'}$. Em que cada lista $\mathcal{L}_i$ é associada a um *coarse centroid* c_i.

Cada entrada de uma lista invertida $\mathcal{L}_i$ correspondente a y contém um identificador do vetor e seu resíduo quantizado $q_p(r(y))$. Dependendo da natureza dos vetores a serem armazenados, o identificador poder não ser único. Imagens com descritores locais podem utilizar o mesmo identificador para descritores correspondentes a mesma imagem.

3.3.3 Busca em Espaços Quantizados

A utilização da lista invertida para evitar a busca exaustiva provém da ideia de verificar apenas os vetores da base que estão assinalados para o mesmo centroide que o vetor de busca. No entanto, o vetor de busca x e seus vizinhos mais próximos na base nem sempre são assinalados para o mesmo centroide, mas para centroides próximos. Para contornar isso, o vetor de consulta x é indexado para os w centroides mais próximos, ao invés de apenas um. A partir disso, todas as listas associadas aos w centroides são verificadas na busca.

A parte superior da Figura 3.2 ilustra o processo de indexação, detalhado no Algoritmo 5. Dado um vetor y_i da base de dados, ele é quantizado (linha 2) e é computado resíduo entre o vetor y_i e seu valor quantizado $q_c(y_i)$ (linha 3). Em seguida é feito o produto da quantização de $r(y_i)$ (linha 4) e um novo item é inserido na entrada da lista invertida correspondente a $q_c(y)$, contendo o identificador do vetor e seu código binário (linha 5).

Algoritmo 5: INDEXAÇÃO DOS DADOS

Entrada: base de dados D
Saída: lista invertida L
1 **para** *cada vetor* $y_i \in D$ **faça**
2 $\quad y_i_quantizado \leftarrow q_c(y_i)$
3 $\quad residuo_y_i \leftarrow y_i - y_i_quantizado$
4 $\quad codigo_y_i \leftarrow q_p(r(y))$
5 $\quad L[coarse_centroid(y_i_quantizado)] \leftarrow \{y_i_id, y_i_codigo\}$
6 **fim**
7 **retorna** L

Na parte inferior da Figura 3.2 é ilustrado o processo de busca, detalhado no Algoritmo 6. Primeiro o vetor é quantizado para os w vizinhos mais próximos do *codebook* de q_c (linha 2), para cada uma das w entradas, é calculado o resíduo do vetor de consulta e a distância entre o vetor residual e todos os vetores dessa entrada da lista invertida (linhas 4 e 5). Por fim são retornados os k vizinhos mais próximos do vetor de consulta (linhas 8 e 9).

Algoritmo 6: BUSCA

Entrada: consultas

Saída: vizinhos mais próximos kNN

```
1 enquanto consultas ≠ ∅ faça
2 |   nn_centroides ← w coarse centroids mais próximos da consulta
3 |   para cada centroide ∈ nn_centroides faça
4 |   |   residuo_consulta ← compute o vetor residual
5 |   |   distancias ← calcule a distância entre o vetor residual e todas os vetores
  |   |     indexados dessa entrada da lista invertida
6 |   fim
7 fim
8 kNN ← k vizinhos mais próximos baseado nas distâncias calculadas
9 retorna kNN
```

Em comparação à busca exaustiva utilizando ADC, essa busca é significantemente mais rápida. A indexação por meio da lista invertida aumenta a computação ao fazer a quantização de x para $q_c(x)$, que consiste no cálculo de k' distâncias entre vetores D-dimensionais. No entanto, assumindo que a lista invertida seja balanceada, cerca de $n \times w/k'$ entradas são verificadas, diminuindo muito o gasto com computação.

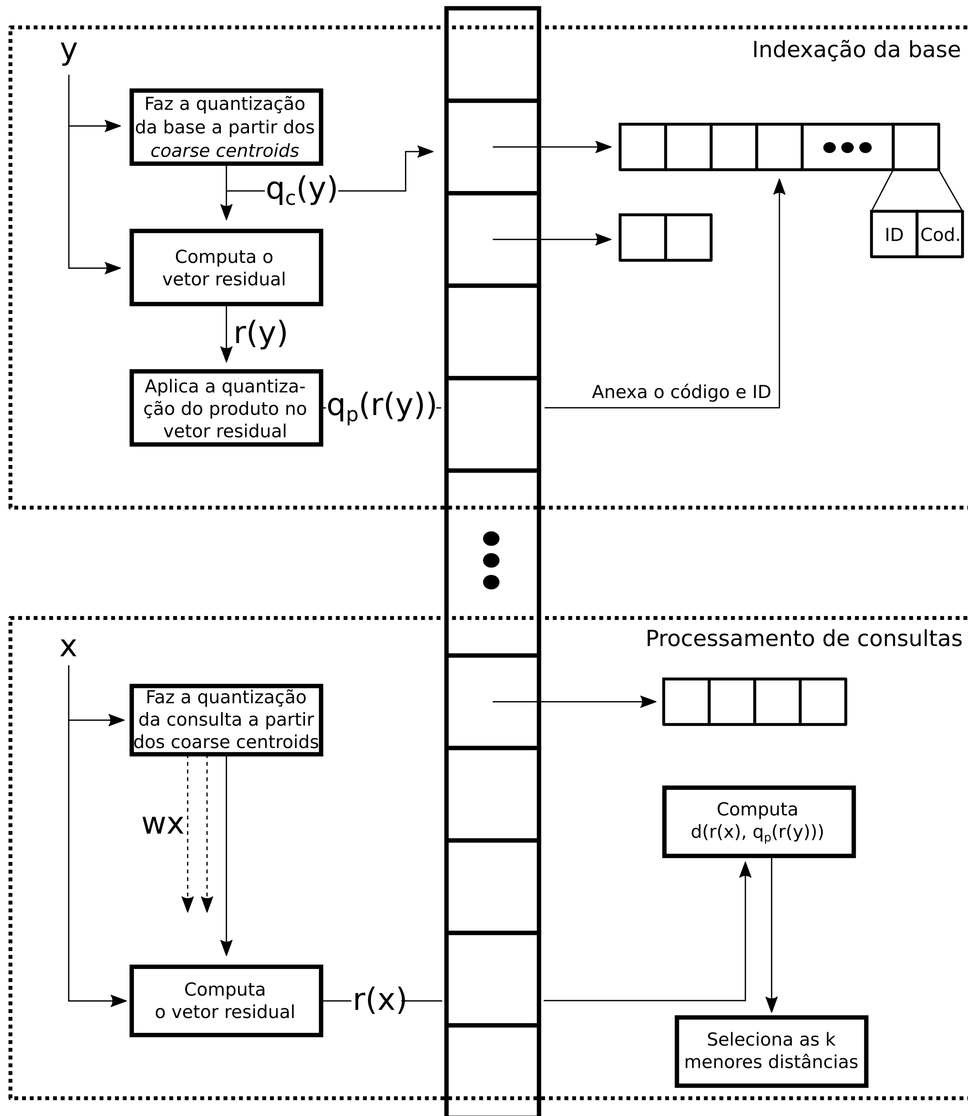

Figura 3.2: Visão geral do sistema de lista invertida com cálculo assimétrico de distância.

Capítulo 4

Paralelização do PQANNS

O algoritmo PQANNS/IVFADC faz um bom compromisso entre qualidade de busca, tempo de execução e uso de memória. No entanto sua implementação sequencial, com apenas uma máquina, não é capaz de suprir as necessidades de um sistema real. Para isso é necessário que o sistema consiga trabalhar com grandes bases de dados, que geralmente não cabem na memória de uma máquina, e minimizar o tempo de resposta de consultas individuais em cenários onde a taxa de chegada de consultas é alta.

Nesse capítulo é detalhada nossa proposta de paralelização desse algoritmo, que permite sua execução em máquinas com memória distribuída e CPUs com vários núcleos. A execução distribuída torna o algoritmo escalável, de forma que trabalhar com bases maiores dependa apenas da disponibilidade de máquinas. Quando aliado ao uso de máquinas multicore, é possível responder uma quantidade maior de consultas mantendo um tempo de resposta baixo.

Na Seção 4.1 será explicada a implementação em memória distribuída, detalhando a decomposição do problema e o fluxo de execução proposto. A Seção 4.2 mostra a paralelização interna do algoritmo, aproveitando a concorrência entre os núcleos para utilizar todo o recurso computacional disponível.

4.1 Paralelização em Memória Distribuída

Há diversas possibilidade de implementação do PQANNS em memória distribuída, sendo que uma delas consiste na replicação da base de dados nas máquinas e a distribuição das consultas entre elas. Dessa forma, cada máquina executa inteiramente uma parte das consultas e retorna diretamente o resultado final. Essa solução escala conforme o aumento do número de consultas, no entanto não consegue lidar com bases de dados maiores que a memória disponível em uma única máquina.

Como a replicação não é uma solução razoável para o tratamento de grandes bases de dados, uma alternativa é o particionamento da base de dados entre as máquinas. No PQANNS, é possível dividir a base a partir dos *coarse centroids*, assim cada máquina armazena os grupos de vetores assinalados para uma quantidade de centroides. Nesse modelo cada consulta é feita em uma quantidade variável de máquinas, o que torna necessário um processo que agregue os resultados locais de cada máquina. Além disso, o balanceamento de carga depende das consultas e não é possível garantir a uniformidade.

A solução adotada no trabalho é o 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 Message Passing Interface (MPI), utilizando a versão 3.1 da implementação da Intel. A aplicação foi decomposta em estágios baseados em fluxo de dados [66, 67, 68, 69, 70], de forma que esses estágios se comuniquem por meio de fluxos direcionados.

A estratégia de paralelização empregada decompõe o PQANNS em quatro estágios: 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 4.1) 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_id), 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 e evitando o desbalanceamento de carga durante o fase de busca.

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

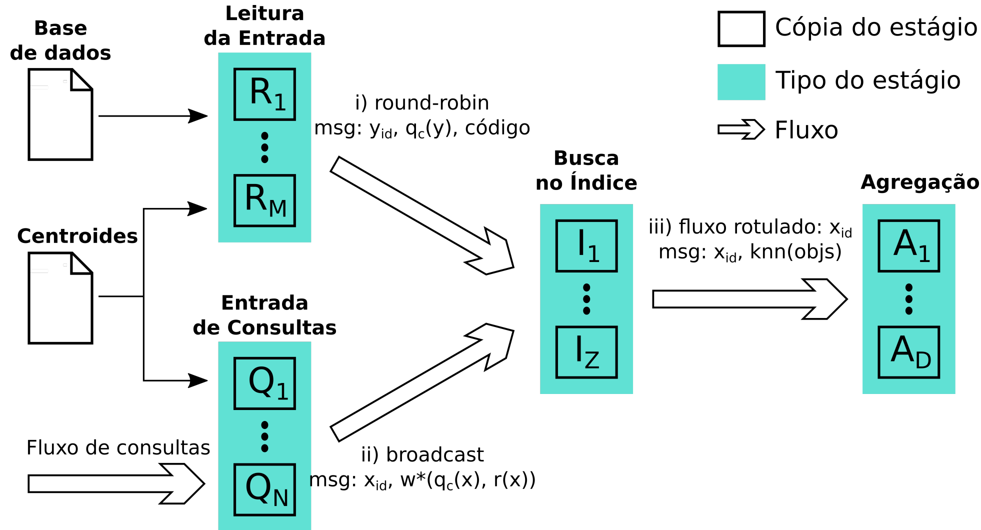

Figura 4.1: Decomposição do PQANNS em um paradigma de fluxo de dados. Na fase de construção do índice, os dados da base são particionados pelas cópias de Leitura da Entrada e enviados para as de Busca no Índice, sem replicação de dados (mensagem i). Nesse estágio são criadas as listas invertidas que indexam os dados locais de cada cópia. Durante a fase de busca, as cópias de Busca no Índice recebem informações sobre o vetor de consulta lido pelas de Entrada de Consultas e computa localmente os NN de sua partição da base de dados de entrada. Em seguida, o estágio de Agregação recebe os resultados locais e computa os NN globais resultantes.

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 meio de *broadcast* (Algoritmo 7). O *broadcast* causa pouco impacto na escalabilidade do fluxo de dados, pois o gasto computacional é dominado pela busca no indice, e a comunicação é feita em segundo plano por meio de *threads* de comunicação.

Algoritmo 7: Entrada de Consultas

```
1 enquanto consultas ≠ ∅ faça
2     consulta ← leia(consultas)
3     consulta_quantizada ← quantize(consulta, w)
4     MPI_Bcast(consulta_quantizada, ...)
5 fim
```

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 (Algoritmo 8).

Algoritmo 8: BUSCA NO ÍNDICE

```
1 consulta ← ∅
2 enquanto verdadeiro faça
3     consulta ← MPI_Recv(...)
4     knn_local ← PQANNS(consulta)
5     dest_Agregacao ← x_id%D
6     MPI_Send(knn_local, dest_Agregacao)
7 fim
```

O estágio de Agregação recebe os dados de todas as cópias do estágio de Busca no Índice por meio de 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 meio de 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 4.1).

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 (Algoritmo 9). 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.

Algoritmo 9: AGREGAÇÃO

```
1 enquanto verdadeiro faça
2     knn ← MPI_Recv(...)
3     reduzir(knn, x_id)
4     contador[x_id] + +
5     se contador[x_id] == Z então
6         saidaGlobalKNN(x_id)
7     fim
8 fim
```

4.2 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. Para implementar essa otimização foi utilizada a biblioteca OpenMP, versão 3.1.

Em busca de reduzir o consumo de memória e permitir a execução concorrente da contrução do índice, o trecho da base a ser indexado pelo estágio de Busca no Índice é dividido em trechos menores, 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 10: 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     $L' \leftarrow indexa(D')$ `//Algoritmo 5`
5   #*pragma* omp critical
6     $L \leftarrow concatena(L')$
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, nela 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 e15), 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).

Algoritmo 11: 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
```

Capítulo 5

Análise dos Resultados

Neste capítulo será apresentada uma análise dos resultados obtidos com as implementações dos algoritmos descritos no Capítulo 3 e no Capítulo 4. As seções a seguir detalham os ambientes de desenvolvimento e testes (Seção 5.1) e os resultados alcançados (Seção 5.2).

5.1 Ambiente de Desenvolvimento e Testes

Os testes foram executados no supercomputador Bridges, uma máquina de memória distribuída constituída por 128 nós interconectados por meio de uma switch FDR Infiniband. Cada nó roda o sistema operacional CentOS 7.4 e segue as seguintes configurações:

- 02 (dois) processadores Intel® Xeon® (E5-2695 v3) 14-core 64-bit E5-processors com 2.3GHz de frequência;
- 35MB de memória cache LLC;
- 128GB de memória RAM DDR4-2133MHz;
- 2 HDDs de 4TB.

Para efeito de comparação, a versão mais recente disponível do FLANN [19] foi utilizada ao lado da nossa implementação PQANNS.

5.2 Resultados Experimentais

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.

5.2.1 Comparação com o Estado da Arte: FLANN

A abordagem para a busca aproximada de vizinhos mais próximos apresentada no FLANN [12, 51, 19] é conhecida por sua eficiência. Ela emprega estruturas hierárquicas (árvores kd, árvores *k-means*) e pode selecionar o algoritmo mais eficiente dentre os disponíveis. Ele também faz o *tuning* dos parâmetros do algoritmo na fase de treinamento para obter a melhor desempenho. 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 [12]. 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. Por uma questão de comparação, o mesmo experimento foi executado para o cálculo exato dos k vizinhos mais próximos, utilizando a biblioteca Yael [71]. Yael é uma biblioteca que implementa de forma altamente otimizada funções utilizadas em aplicações de recuperação de imagens, como clusterização, listas invertidas e busca dos vizinhos mais próximos. Para a realização da busca exata, Yael utiliza funções das bibliotecas Basic Linear Algebra Subprograms (BLAS) e Linear Algebra Package (LAPACK),

conhecidas por sua eficiência. Para referência, utilizando a Yael, o experimento levou cerca de 212 segundos.

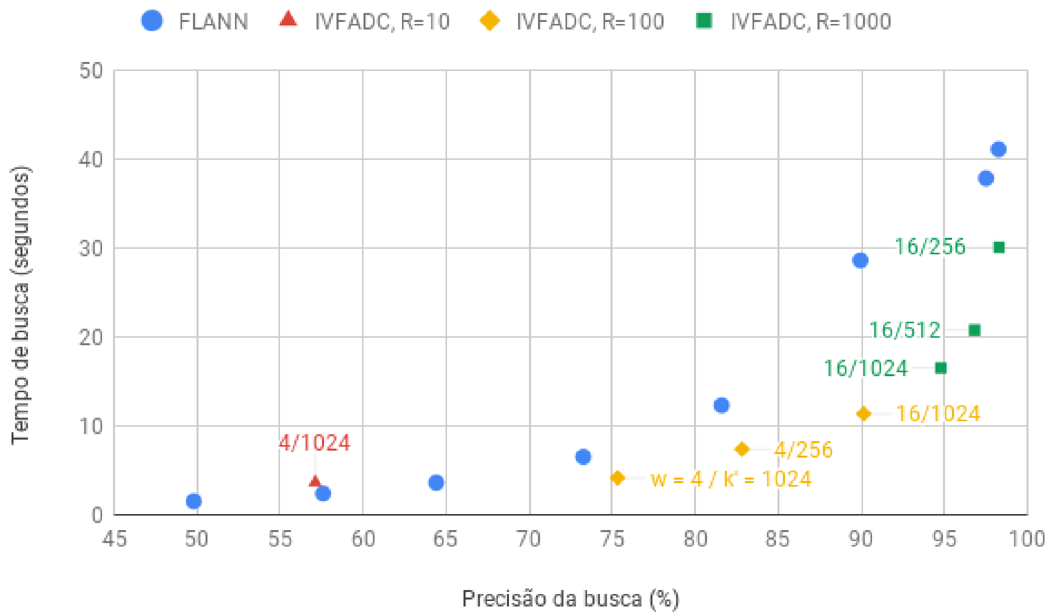

Figura 5.1: IVFADC vs FLANN: compromissos entre qualidade da busca (precisão) e tempo de busca..

O resultado experimental comparando os tempos de busca do IVFADC/PQANNS e do FLANN enquanto a precisão é variada é mostrado na Figura 5.1. 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. No entanto, o FLANN mantém sua vantagem de fazer o *tuning* automático de parâmetros, sendo necessário apenas informar a precisão de busca desejada e os pesos referentes à importância do tempo de busca e uso de memória. Já no PQANNS foi necessária a execução de diversos testes variando os parâmetros até atingir os melhores resultados para uma dada precisão.

Em relação ao uso de memória, o PQANNS utiliza cerca de 25 MB de RAM para realizar buscas em uma base com 1 milhão de vetores, o que corresponde ao tamanho da lista invertida, contendo apenas os índices e códigos dos descritores. Já o FLANN requer mais de 600 MB de RAM para realizar as mesmas consultas, pois ele necessita manter os descritores em memória para realizar um rerranqueamento dos resultados, em busca de retornar com mais precisão os vizinhos mais próximos.

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. Já as vantagens do FLANN são a opção de *tuning* automático, que

facilita o uso do algoritmo, e o rerranqueamento dos resultados, o que garante a precisão desejada para o retorno de um vetor, enquanto o PQANNS garante a precisão dentre R vetores retornados na busca. Vale notar que existem implementações de rerranqueamento no PQANNS, no entanto elas também incrementam o uso de memória.

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, é de 212 segundos, para uma precisão de 98% o PQANNS e o FLANN demoram 31 segundos e 40 segundos, respectivamente.

5.2.2 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, verificando 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.

Na Figura 5.2 é mostrada a variação do *speedup* sobre o algoritmo sequencial atingido pela aplicação em diversos níveis de precisão, enquanto é variado o número de *threads*. Pode-se notar que o desempenho geral melhora quando é aumentado o número de *threads*.

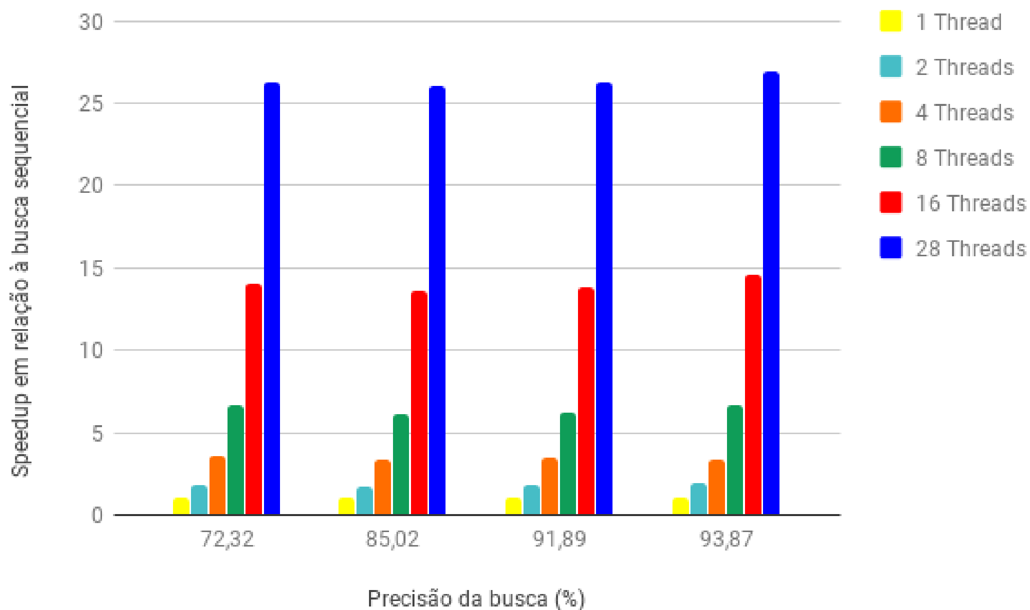

Figura 5.2: Variação do *speedup* sobre o algoritmo sequencial em relação à variação do número de *threads*.

A obtenção de *speedups* próximos ao linear mostra a capacidade de escalar do algoritmo, que em todas as configurações consegue utilizar todos os núcleos disponíveis de forma eficiente, sem transferir o gargalo para outro recurso computacional.

O baixo consumo de memória da estrutura de lista invertida e o bom gerenciamento de concorrência utilizando OpenMP faz com que a memória não seja um gargalo quando são utilizados muitos núcleos. Os testes mostram que nossa aplicação ganha desempenho com até 28 núcleos.

5.2.3 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 5.3 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. O tráfego de rede da comunicação do sistema conforme o número de nós muda é apresentado na Figura 5.4. Como mostrado, o tráfego na rede é muito baixo apesar do número de nós utilizados, o que é outro fator promissor da solução.

Diferentemente de outras paralelizações, como as do algoritmo LSH [17, 18], a nossa preserva o comportamento do algoritmo sequencial, sem fazer replicações da base ou impor limitações de comunicação. O baixo tráfego de rede indica que o algoritmo continuará escalável se uma quantidade muito maior de nós for utilizada, superando a limitação identificada na paralelização do FLANN proposta em [19].

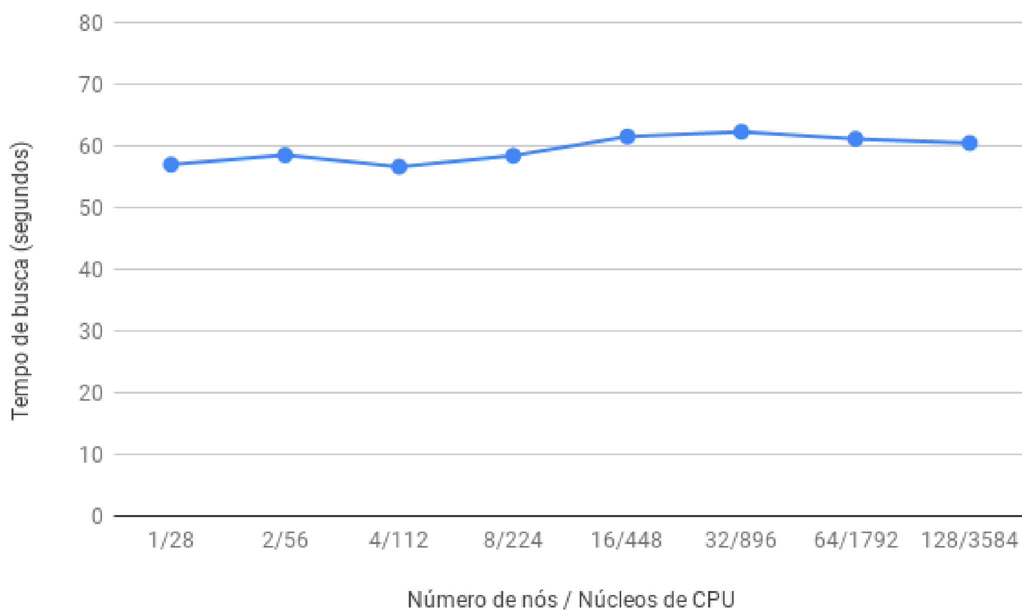

Figura 5.3: Escalabilidade da paralelização em memória distribuída em um experimento de escalabilidade fraca utilizando uma base de dados contendo 256 bilhões de descritores SIFT na configuração com 128 nós.

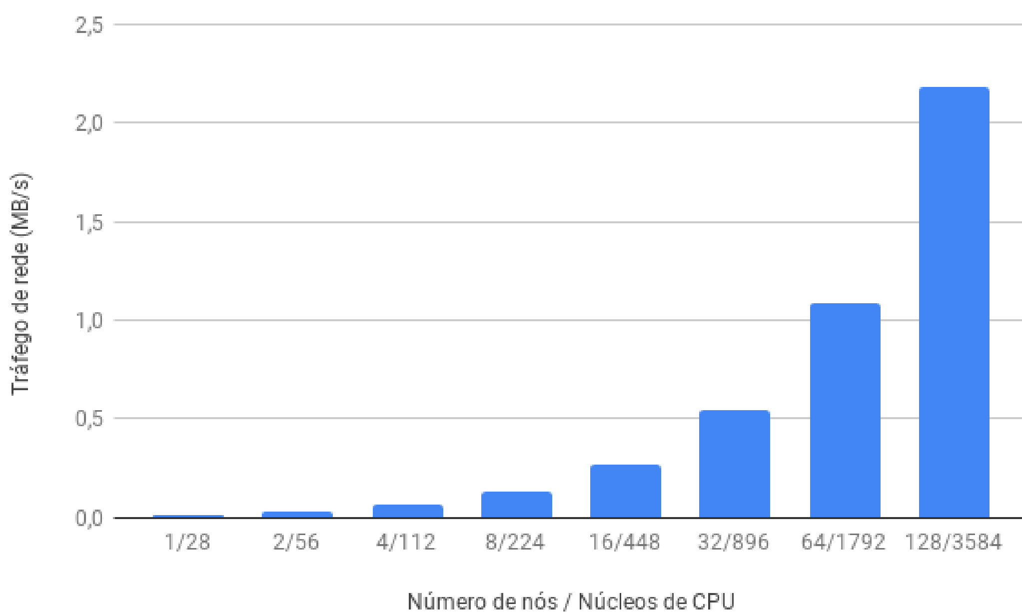

Figura 5.4: Tráfego de rede (MB/s) em relação à variação do número de nós em um experimento de escalabilidade fraca e uma base de dados contendo 256 bilhões de descritores SIFT na configuração com 128 nós.

Capítulo 6

Conclusão

Esse trabalho de graduação 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. Em muitos casos, a busca exata não é um requisito para a aplicação e pode ser substituída por uma aproximação mais eficiente.

Nós utilizamos o algoritmo PQANNS, que faz a redução da dimensionalidade do espaço de busca através da quantização de espaços, para isso são construídos códigos através da clusterização de subdimensões dos vetores da base, que aproximam os vetores originais. Além disso, é utilizada uma estrutura de indexação de lista invertida, que guarda os códigos quantizados dos vetores da base de dados e reúne os mais próximos na mesma entrada da lista. Isso permita a realização de uma busca não-exaustiva na base, onde apenas as listas mais próximas do vetor de consulta são buscadas.

A partir dessas otimizações, 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 distribuição entre máquinas foi feita em cima da interface MPI, decompondo a aplicação em estágios baseados em fluxo de dados. São quatro estágios (Leitura da Entrada, Entrada de Consultas, Busca no Índice e Agregação), conectados por dois fluxos de dados. O primeiro fluxo de dados consiste na construção do índice, para isso os dados da base são particionados pelas cópias de Leitura da Entrada e enviados para as de Busca no Índice, sem replicação de dados. Nesse estágio são criadas as listas invertidas que indexam os dados locais de cada cópia. O segundo fluxo é responsável pela busca, em que as cópias de Busca no Índice recebem informações sobre o vetor de consulta lido pelas de Entrada de Consultas e computa localmente os NN de sua partição da base de dados de entrada.

Em seguida, o estágio de Agregação recebe os resultados locais e computa os NN globais resultantes.

Essa paralelização permite que o algoritmo realize busca em bases muito maiores, sem as limitações de memória de apenas uma máquina. As avaliações do Capítulo 5 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 por meio da troca de mensagens.

A paralelização interna, utilizando OpenMP, permite um melhor uso do poder de processamento disponível. Para isso foram paralelizados os processos que fazem a construção do índice e a busca, os mais caros da aplicação. Dessa forma, o trabalho foi dividido entre os núcleos da máquina e houve uma melhoria considerável no desempenho do algoritmo.

As avaliações realizadas no Capítulo 5 mostraram as vantagens do PQANNS ante o FLANN, que obteve um melhor compromisso entre precisão e tempo de resposta. Foram realizados testes de escalabilidade horizontal, que mostraram a capacidade da nossa paralelização de trabalhar com diversas máquinas sem perder desempenho, realizando buscas na maior base disponível na literatura. Por fim, nossa paralelização interna atingiu ótimos *speedups* em diferentes configurações do algoritmo, atingindo melhoria de desempenho em todos eles.

6.1 Trabalhos Futuros

A paralelização desenvolvida nesse trabalho lida com consultas em *batch* e com uma base de dados estática, o que não corresponde a uma aplicação real, em que o volume de consultas e os dados da base variam com o tempo. Como trabalho futuro, deseja-se propor uma uma solução que trabalhe com *streaming* de dados, atualizando a base em tempo real e que lide com cargas variáveis.

Além disso, é possível integrar a arquitetura proposta com a utilização de GPUs para acelerar o processamento das consultas. Uma solução heterogênea pode melhorar os tempos de resposta da aplicação e fazer uma utilização mais inteligente de recursos computacionais. Pretendemos para tanto utilizar e adaptar técnicas que empregamos em outros domínios [72, 73, 74, 75, 76].

Por fim, é importante avaliar a utilização de formas de indexação mais complexas associadas à quantização do produto, bem como formas de reranqueamento dos dados que garantam um aumento na qualidade da busca.

Referências

[1] Constine, Josh: *Facebook hits 100m hours of video watched a day, 1b users on groups, 80m on fb lite.* `https://techcrunch.com/2016/01/27/facebook-grows/`. Accessed: 2018-11-04. 1

[2] Alba, Davey: *Instagram now tops 400 million users and 40 billion photos.* `https://www.wired.com/2015/09/instagram-now-tops-400-million-users-40-billion-photos/`. Accessed: 2018-11-04. 1

[3] Dredge, Stuart: *Instagram now has 300m users sharing 70m photos and videos a day.* `https://www.theguardian.com/technology/2014/dec/10/instagram-300m-users-70m-photos-videos`. Accessed: 2018-11-04. 1

[4] Etherington, Darrell: *People now watch 1 billion hours of youtube per day.* `https://techcrunch.com/2017/02/28/people-now-watch-1-billion-hours-of-youtube-per-day/`. Accessed: 2018-11-04. 1

[5] Lew, Michael S, Nicu Sebe, Chabane Djeraba e Ramesh Jain: *Content-based multimedia information retrieval: State of the art and challenges.* ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM), 2(1):1–19, 2006. 1

[6] Jain, Mihir: *Enhanced image and video representation for visual recognition.* Tese de Doutoramento, Université Rennes 1, 2014. 1, 2, 9

[7] Sawhney, Harpreet S e James L Hafner: *Efficient color histogram indexing.* Em *Image Processing, 1994. Proceedings. ICIP-94., IEEE International Conference*, volume 2, páginas 66–70. IEEE, 1994. 2

[8] Mumford, David: *The problem of robust shape descriptors.* Center for Intelligent Control Systems, 1987. 2

[9] Jagadish, Hosagrahar V: *A retrieval technique for similar shapes.* ACM SIGMOD Record, 20(2):208–217, 1991. 2

[10] Wallace, Timothy P e Paul A Wintz: *An efficient three-dimensional aircraft recognition algorithm using normalized fourier descriptors.* Computer Graphics and Image Processing, 13(2):99–126, 1980. 2

[11] Friedman, Jerome H, Jon Louis Bentley e Raphael Ari Finkel: *An algorithm for finding best matches in logarithmic expected time.* ACM Transactions on Mathematical Software (TOMS), 3(3):209–226, 1977. 2, 12, 14

[12] Muja, Marius e David G Lowe: *Fast approximate nearest neighbors with automatic algorithm configuration.* VISAPP (1), 2(331-340):2, 2009. 2, 12, 13, 41

[13] Böhm, Christian, Stefan Berchtold e Daniel A Keim: *Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases.* ACM Computing Surveys (CSUR), 33(3):322–373, 2001. 2

[14] Datar, Mayur, Nicole Immorlica, Piotr Indyk e Vahab S Mirrokni: *Locality-sensitive hashing scheme based on p-stable distributions.* Em *Proceedings of the twentieth annual symposium on Computational geometry*, páginas 253–262. ACM, 2004. 2

[15] Gionis, Aristides, Piotr Indyk, Rajeev Motwani *et al.*: *Similarity search in high dimensions via hashing.* Em *Vldb*, volume 99, páginas 518–529, 1999. 2, 13, 24

[16] Jegou, Herve, Matthijs Douze e Cordelia Schmid: *Product quantization for nearest neighbor search.* IEEE transactions on pattern analysis and machine intelligence, 33(1):117–128, 2011. 3, 13, 20, 21, 28, 29

[17] Stupar, Aleksandar, Sebastian Michel e Ralf Schenkel: *Rankreduce-processing k-nearest neighbor queries on top of mapreduce.* Large-Scale Distributed Systems for Information Retrieval, 15, 2010. 3, 24, 44

[18] Bahmani, Bahman, Ashish Goel e Rajendra Shinde: *Efficient distributed locality sensitive hashing.* Em *Proceedings of the 21st ACM international conference on Information and knowledge management*, páginas 2174–2178. ACM, 2012. 3, 24, 44

[19] Muja, Marius e David G Lowe: *Scalable nearest neighbor algorithms for high dimensional data.* IEEE Transactions on Pattern Analysis & Machine Intelligence, (11):2227–2240, 2014. 3, 13, 24, 25, 40, 41, 44

[20] Johnson, Jeff, Matthijs Douze e Hervé Jégou: *Billion-scale similarity search with gpus.* arXiv preprint arXiv:1702.08734, 2017. 3, 24

[21] Wieschollek, Patrick, Oliver Wang, Alexander Sorkine-Hornung e Hendrik Lensch: *Efficient large-scale approximate nearest neighbor search on the gpu.* Em *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, páginas 2027–2035, 2016. 3, 24

[22] Andrade, Guilherme, André Fernandes, Jeremias M Gomes, Renato Ferreira e George Teodoro: *Large-scale parallel similarity search with product quantization for online multimedia services.* Journal of Parallel and Distributed Computing, 125:81–92, 2019. 4

[23] Mikolajczyk, Krystian e Cordelia Schmid: *Scale & affine invariant interest point detectors.* International journal of computer vision, 60(1):63–86, 2004. 6

[24] Lowe, David G: *Distinctive image features from scale-invariant keypoints.* International journal of computer vision, 60(2):91–110, 2004. 6, 7, 8

[25] Mikolajczyk, Krystian, Tinne Tuytelaars, Cordelia Schmid, Andrew Zisserman, Jiri Matas, Frederik Schaffalitzky, Timor Kadir e Luc Van Gool: *A comparison of affine region detectors.* International journal of computer vision, 65(1-2):43–72, 2005. 7

[26] Bay, Herbert, Tinne Tuytelaars e Luc Van Gool: *Surf speeded up robust features.* Em *European conference on computer vision*, páginas 404–417. Springer, 2006. 7

[27] Dalal, Navneet e Bill Triggs: *Histograms of oriented gradients for human detection.* Em *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, páginas 886–893. IEEE, 2005. 7

[28] Jegou, Herve, Matthijs Douze e Cordelia Schmid: *Hamming embedding and weak geometric consistency for large scale image search.* Em *European conference on computer vision*, páginas 304–317. Springer, 2008. 8

[29] Boiman, Oren, Eli Shechtman e Michal Irani: *In defense of nearest-neighbor based image classification.* Em *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, páginas 1–8. IEEE, 2008. 8

[30] Sivic, Josef e Andrew Zisserman: *Video google: A text retrieval approach to object matching in videos.* Em *null*, página 1470. IEEE, 2003. 9

[31] Csurka, Gabriella, Christopher Dance, Lixin Fan, Jutta Willamowski e Cédric Bray: *Visual categorization with bags of keypoints.* Em *Workshop on statistical learning in computer vision, ECCV*, volume 1, páginas 1–2. Prague, 2004. 9

[32] Jégou, Hervé, Matthijs Douze, Cordelia Schmid e Patrick Pérez: *Aggregating local descriptors into a compact image representation.* Em *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, páginas 3304–3311. IEEE, 2010. 9, 10

[33] Perronnin, Florent e Christopher Dance: *Fisher kernels on visual vocabularies for image categorization.* Em *2007 IEEE conference on computer vision and pattern recognition*, páginas 1–8. IEEE, 2007. 9

[34] Perronnin, Florent, Christopher Dance, Gabriela Csurka e Marco Bressan: *Adapted vocabularies for generic visual categorization.* Em *European Conference on Computer Vision*, páginas 464–475. Springer, 2006. 9

[35] Philbin, James, Ondrej Chum, Michael Isard, Josef Sivic e Andrew Zisserman: *Lost in quantization: Improving particular object retrieval in large scale image databases.* 2008. 9

[36] Van Gemert, Jan C, Cor J Veenman, Arnold WM Smeulders e Jan Mark Geusebroek: *Visual word ambiguity.* IEEE Transactions on Pattern Analysis & Machine Intelligence, (7):1271–1283, 2009. 9

[37] Zhou, Xi, Kai Yu, Tong Zhang e Thomas S Huang: *Image classification using super-vector coding of local image descriptors.* Em *European conference on computer vision*, páginas 141–154. Springer, 2010. 9

[38] Avila, Sandra, Nicolas Thome, Matthieu Cord, Eduardo Valle e A de A Araújo: *Bossa: Extended bow formalism for image classification.* Em *Image Processing (ICIP), 2011 18th IEEE International Conference on*, páginas 2909–2912. IEEE, 2011. 10

[39] Jégou, Hervé e Ondřej Chum: *Negative evidences and co-occurences in image retrieval: The benefit of pca and whitening.* Em *Computer Vision–ECCV 2012*, páginas 774–787. Springer, 2012. 10

[40] Negrel, Romain, David Picard e Philippe Henri Gosselin: *Using spatial pyramids with compacted vlat for image categorization.* Em *Pattern Recognition (ICPR), 2012 21st International Conference on*, páginas 2460–2463. IEEE, 2012. 10

[41] Oliva, Aude e Antonio Torralba: *Modeling the shape of the scene: A holistic representation of the spatial envelope.* International journal of computer vision, 42(3):145–175, 2001. 11

[42] Douze, Matthijs, Hervé Jégou, Harsimrat Sandhawalia, Laurent Amsaleg e Cordelia Schmid: *Evaluation of gist descriptors for web-scale image search.* Em *Proceedings of the ACM International Conference on Image and Video Retrieval*, página 19. ACM, 2009. 11

[43] Chatzichristofis, Savvas A e Yiannis S Boutalis: *Fcth: Fuzzy color and texture histogram-a low level feature for accurate image retrieval.* Em *Image Analysis for Multimedia Interactive Services, 2008. WIAMIS'08. Ninth International Workshop on*, páginas 191–196. IEEE, 2008. 11

[44] Wang, Hai e Shuwu Zhang: *Evaluation of global descriptors for large scale image retrieval.* Em *International Conference on Image Analysis and Processing*, páginas 626–635. Springer, 2011. 11

[45] Siagian, Christian e Laurent Itti: *Rapid biologically-inspired scene classification using features shared with visual attention.* IEEE transactions on pattern analysis and machine intelligence, 29(2):300–312, 2007. 11

[46] Faloutsos, Christos, Ron Barber, Myron Flickner, Jim Hafner, Wayne Niblack, Dragutin Petkovic e William Equitz: *Efficient and effective querying by image content.* Journal of intelligent information systems, 3(3-4):231–262, 1994. 11

[47] Beis, Jeffrey S e David G Lowe: *Shape indexing using approximate nearest-neighbour search in high-dimensional spaces.* Em *cvpr*, página 1000. IEEE, 1997. 12, 13

[48] Silpa-Anan, Chanop e Richard Hartley: *Optimised kd-trees for fast image descriptor matching.* Em *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, páginas 1–8. IEEE, 2008. 12, 14

[49] Fukunaga, Keinosuke e Patrenahalli M. Narendra: *A branch and bound algorithm for computing k-nearest neighbors*. IEEE transactions on computers, 100(7):750–753, 1975. 12, 13

[50] Beygelzimer, Alina, Sham Kakade e John Langford: *Cover trees for nearest neighbor*. Em *Proceedings of the 23rd international conference on Machine learning*, páginas 97–104. ACM, 2006. 12

[51] Muja, Marius e David G Lowe: *Fast matching of binary features*. Em *Computer and Robot Vision (CRV), 2012 Ninth Conference on*, páginas 404–410. IEEE, 2012. 13, 41

[52] Muja, Marius: *Scalable nearest neighbour methods for high dimensional data*. Tese de Doutoramento, University of British Columbia, 2013. 13, 17

[53] Valle, Eduardo, Matthieu Cord e Sylvie Philipp-Foliguet: *High-dimensional descriptor indexing for large multimedia databases*. Em *Proceedings of the 17th ACM conference on Information and knowledge management*, páginas 739–748. ACM, 2008. 13, 20

[54] Nelder, John A e Roger Mead: *A simplex method for function minimization*. The computer journal, 7(4):308–313, 1965. 18

[55] Indyk, Piotr e Rajeev Motwani: *Approximate nearest neighbors: towards removing the curse of dimensionality*. Em *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, páginas 604–613. ACM, 1998. 19

[56] Faloutsos, Christos: *Multiattribute hashing using gray codes*. Em *ACM SIGMOD Record*, volume 15, páginas 227–238. ACM, 1986. 19

[57] Skubalska-Rafajlowicz, Ewa e Adam Krzyzak: *Fast k-nn classification rule using metric on space-filling curves*. Em *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 2, páginas 121–125. IEEE, 1996. 19

[58] Teodoro, George, Eduardo Valle, Nathan Mariano, Ricardo Torres, Jr Meira, Wagner e JoelH. Saltz: *Approximate similarity search for online multimedia services on distributed CPU–GPU platforms*. The VLDB Journal, páginas 1–22, 2013, ISSN 1066-8888. `http://dx.doi.org/10.1007/s00778-013-0329-7`. 20

[59] Teodoro, George, Eduardo Valle, Nathan Mariano, Ricardo da Silva Torres e Wagner Meira Jr.: *Adaptive parallel approximate similarity search for responsive multimedia retrieval*. Em *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, páginas 495–504, 2011. 20

[60] Duncan, Ralph: *A survey of parallel computer architectures*. Computer, (2):5–16, 1990. 21, 22

[61] Flynn, Michael J: *Some computer organizations and their effectiveness*. IEEE transactions on computers, 100(9):948–960, 1972. 21

[62] Moise, Diana, Denis Shestakov, Gylfi Gudmundsson e Laurent Amsaleg: *Indexing and searching 100m images with map-reduce*. Em *Proceedings of the 3rd ACM conference on International conference on multimedia retrieval*, páginas 17–24. ACM, 2013. 24

[63] Andrade, Guilherme, George Teodoro e Renato Ferreira: *Online Multimedia Similarity Search with Response Time-Aware Parallelism and Task Granularity Auto-Tuning*. Em *29th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2017, Campinas, Brazil, October 17-20, 2017*, páginas 153–160, 2017. 24

[64] Teixeira, Thiago S. F. X., George Teodoro, Eduardo Valle e Joel H. Saltz: *Scalable Locality-Sensitive Hashing for Similarity Search in High-Dimensional, Large-Scale Multimedia Datasets*. CoRR, abs/1310.4136:1–20, 2013. `http://arxiv.org/abs/1310.4136`. 24

[65] Silva, Eliezer, Thiago Teixeira, George Teodoro e Eduardo Valle: *Large-scale distributed locality-sensitive hashing for general metric data*. Em Traina, Agma Juci Machado, Caetano Traina e Robson Leonardo Ferreira Cordeiro (editores): *Similarity Search and Applications*, páginas 82–93. Springer International Publishing, 2014. 24

[66] Arpaci-Dusseau, Remzi H, Eric Anderson, Noah Treuhaft, David E Culler, Joseph M Hellerstein, David Patterson e Kathy Yelick: *Cluster i/o with river: Making the fast case common*. Em *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, páginas 10–22. ACM, 1999. 35

[67] Beynon, Michael D, Tahsin Kurc, Umit Catalyurek, Chialin Chang, Alan Sussman e Joel Saltz: *Distributed processing of very large datasets with datacutter*. Parallel Computing, 27(11):1457–1478, 2001. 35

[68] Teodoro, George, Daniel Fireman, Dorgival Guedes, Wagner Meira Jr e Renato Ferreira: *Achieving multi-level parallelism in the filter-labeled stream programming model*. Em *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, páginas 287–294. IEEE, 2008. 35

[69] Ferreira, Renato, Wagner Meira Jr., Dorgival Olavo Guedes Neto, Lucia Maria de A. Drummond, Bruno Coutinho, George Teodoro, Tulio Tavares, Renata Braga Araujo e Guilherme T. Ferreira: *Anthill: A scalable run-time environment for data mining applications*. Em *17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) 2005, 24-27 October 2005, Rio de Janeiro, Brazil*, páginas 159–167, 2005. 35

[70] Teodoro, George, Rafael Sachetto Oliveira, Olcay Sertel, Metin N. Gurcan, Wagner Meira Jr., Ümit V. Çatalyürek e Renato Ferreira: *Coordinating the use of GPU and CPU for improving performance of compute intensive applications*. Em *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*, páginas 1–10, 2009. 35

[71] Douze, Matthijs e Hervé Jégou: *The yael library*. Em *Proceedings of the 22nd ACM international conference on Multimedia*, páginas 687–690. ACM, 2014. 41

[72] Teodoro, George, Tahsin M. Kurç, Jun Kong, Lee A. D. Cooper e Joel H. Saltz: *Comparative performance analysis of intel (R) xeon phi (tm), gpu, and CPU: A case study from microscopy image analysis.* Em *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, páginas 1063–1072, 2014. 47

[73] Teodoro, George, Tahsin M. Kurç, Guilherme Andrade, Jun Kong, Renato Ferreira e Joel H. Saltz: *Application performance analysis and efficient execution on systems with multi-core CPUs, GPUs and MICs: a case study with microscopy image analysis.* IJHPCA, 31(1):32–51, 2017. 47

[74] Teodoro, George, Tony Pan, Tahsin M. Kurç, Jun Kong, Lee A. D. Cooper, Scott Klasky e Joel H. Saltz: *Region templates: Data representation and management for high-throughput image analysis.* Parallel Computing, 40(10):589–610, 2014. 47

[75] Teodoro, George, Tony Pan, Tahsin M. Kurç, Jun Kong, Lee A. D. Cooper, Norbert Podhorszki, Scott Klasky e Joel H. Saltz: *High-throughput analysis of large microscopy image datasets on CPU-GPU cluster platforms.* Em *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, páginas 103–114, 2013. 47

[76] Teodoro, George, Timothy D. R. Hartley, Ümit V. Çatalyürek e Renato Ferreira: *Run-time optimizations for replicated dataflows on heterogeneous environments.* Em *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010, Chicago, Illinois, USA, June 21-25, 2010*, páginas 13–24, 2010. 47