

# Learning with Massive Data Arguments Grid

CA' FOSCARI UNIVERSITY OF VENICE  
Department of Environmental Sciences, Informatics and Statistics



Academic Year 2022 - 2023

**Student** Zuliani Riccardo 875532

# Contents

<b>I</b>	<b>Lucchese Part</b>	<b>1</b>
1	Introduction	2
2	Cache Awareness	3
2.1	Cache-to-memory Coherence . . . . .	3
2.2	Matrix Multiplication . . . . .	4
2.3	Sorting & K-Funnel . . . . .	5
2.4	Multi-core Hierarchies: key challenge . . . . .	6
3	Shared Memory: Threads	7
3.1	Performance Metrics . . . . .	7
4	OpenMP & Cache	9
5	Pattern of Parallelism I	10
5.1	Decomposition . . . . .	10
5.1.1	Task Dependency Graph . . . . .	10
5.1.2	Parallelism Degree and Critical Path . . . . .	11
5.1.3	Task Interaction Graph . . . . .	11
5.2	Mapping . . . . .	11
5.3	Prefix-Sum . . . . .	11
6	Pattern of Parallelism II - Common Patterns	13
6.1	Parallelizing Quicksort . . . . .	14
6.1.1	Sequential Version . . . . .	14
6.1.2	Parallel Version . . . . .	14
6.2	Odd-Even Transposition . . . . .	15
6.3	Bitonic Sort . . . . .	15
7	Large-Scale Data Processing with Map-Reduce	17
8	Vertex Centric Paradigm	19
8.1	Label Propagation . . . . .	19
8.2	Hash-To-Min . . . . .	19
<b>II</b>	<b>Bruch Part</b>	<b>20</b>
9	Ranking	21

---

<b>10 Ranking Loss Function</b>	<b>22</b>
<b>11 Representation and Hypothesis Classes</b>	<b>23</b>
<b>12 Complexity of Ranking Functions</b>	<b>25</b>
<b>13 Retrieval with MIPS: Representation Learning</b>	<b>26</b>
13.1 Sparse Representation . . . . .	26
13.2 Dense Representation . . . . .	27
<b>14 Retrieval with MIPS: Sparse Vectors</b>	<b>28</b>
<b>15 Retrieval with MIPS: Dense Vectors</b>	<b>29</b>

**Part I**

**Lucchese Part**

# Chapter 1

## Introduction

- The Moore's Law
- Some advantages of multicores:
  - Power
  - Design cost
  - Defect tolerance
- Sequential VS Parallel computing
  - Sequential computing
  - Parallel Computing
- Clusters
- When is Parallelism Necessary?

# Chapter 2

## Cache Awareness

- Effect of memory latency
- Cache Memory
  - Definition
  - Performance improve in presence of high locality
    - \* Temporal locality
    - \* Spatial locality
  - Cache hit rateo
  - Other approaches for hiding memory latency
    - \* Multi-threading
    - \* Pre-fetching
    - \* Drawbacks (Bandwidth and cache pollution)

### 2.1 Cache-to-memory Coherence

- After updating/writing data in cache, when to write to memory?
- **Write-Through Policy** definition
- **Write-Back Policy** definition, in case of write data not being in cache
  - Write Allocate
  - Write No Allocate
- **Cache-to-cache Coherence in Symmetric Multi-Processors**
  - Snooping Protocols
  - Cache controller snoop all the bus transactions
    - \* Transaction relevant
    - \* Actions to guarantee **cache coherence**
      - *Update*
      - *Invalidate*

- **Update**
  - \* *Pros:*
    - Multiple r/w copies are kept coherent after each write, save bandwidth
  - \* *Con:* Unnecessary waste
    - Cache block is update but no longer read
    - Subsequent writes by the same processor cause multiple updates
- **Invalidate**
  - \* *Pro:*
    - Multiple writes by the same processor do not cause any additional overhead
  - \* *Con:*
    - An access that follows an invalidation causes a miss
- **False Sharing**
  - \* Coherent protocols works in term of cache blocks/lines rather single words/bytes
  - \* Blocks plays an important role in coherence protocol
    - Small blocks the protocols are more efficient
    - Large blocks are better for spartial locality
- **We want to minimize the cache miss and maximize the cache hit**
- External memory model
  - Transfer occur in blocks of size  $B$
  - The cache has size  $M \geq B$
  - With  $M/B$  entries
- **Example of linear scan:**  $\lceil \frac{N}{B} + 1 \rceil$  memory scan

## 2.2 Matrix Multiplication

- Rows in row-wise
- Columns in column-wise
- Each element of matrix  $C$  involves  $O(1 + \frac{N}{B})$  transfers
- Since there are  $N^2$  elements of  $C$  the complexity is  $O(N^2 \frac{N^3}{B})$
- Approach to reduce the cost
- **Improved Algorithm** complexity  $O\left(\frac{N^2}{B} + \frac{N^3}{B \cdot \sqrt{M}}\right)$  with block matrices
- **Cache Oblivious Algorithm**
  - Divide-and-conquer approach

- We recursively split the matrix until at some point the matrix will fit in cache, whatever is its size
- *Recursive Data Layout*, st however we recursively split the matrix, at some point the data will be in consecutive memory location that can be easily loaded in cache
- **Z-Order**
- **Complexity**
  - *Case base*: three blocks fit in cache, since successive recursion step do not cause additional misses
    - \* Block size  $k\sqrt{M} \cdot k\sqrt{M}$  for some constant  $k$
  - Three of such blocks fill the cache with  $M/B$  misses
  - Computational complexity of the block-based multiplication is  $\left(\frac{N}{k\sqrt{M}}\right)^3$
  - Number of misses:  $\left(\frac{N}{k\sqrt{M}}\right)^3 \cdot \frac{M}{B} = O\left(\frac{N^3}{B\sqrt{M}}\right)$
  - When  $N$  is small, data loading:  $O\left(\frac{N^2}{B}\right)$
  - Total cost:  $O\left(\frac{N^2}{B} + \frac{N^3}{B\sqrt{M}}\right)$

## 2.3 Sorting & K-Funnel

- Merge sort
- **Cache-aware solution**
  - M/B-way merge
  - $\theta\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right)$
- **First Cache Oblivious Solution**
  - 2-way merge sort
  - $\theta\left(\frac{N}{B} \log_2 \frac{N}{M}\right)$
  - Can we rise the base of the logarithm to M/M
- **Cache Oblivious Sort: K-Funnel**
  - Definition
  - Description
  - 
  - $\theta\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}\right)$  like cache-aware solution



## 2.4 Multi-core Hierarchies: key challenge

- Good performance for any  $M$  &  $B$  on 2 levels **does not** imply *good performance* at all levels of hierarchy
- Reason: cache is not fully shared, what is **good** for  $cpu_1$  is often **bad** for  $cpu_2$  and  $cpu_3$
- Scheduling of parallel thread has a large impact on cache performance

# Chapter 3

## Shared Memory: Threads

### 3.1 Performance Metrics

- two metrics:
  - Parallel run time
  - Cost
- *When a parallel algorithm is cost optimal*
- **SpeedUp**
  - Formula
  - Typical success
  - Linear speedup
  - Super linear speedup
  - Using more thread require more overhead
- **Efficiency**
  - Measure of resource usage
  - fraction of time the processors are fully used to execute a fraction  $1/p$  of the best sequential algorithm
  - Formula
- **Amdahl's Law (SpeedUp)**
  - Parallel execution time  $T_p$  cannot be arbitrary reduced by increasing  $p$
  - Suppose a fraction  $f$  cannot be executed in parallel
  - Formula
  - The sequential fraction  $f$  is an upper bound on the speedup  $S$
- **Gustafson's Law (Scalability)**
  - We can't use a large number of processors in order to process large amount of data

- Sequential part  $c$  is constant wrt  $n$
- $T(n, p)$  the execution time of the parallelizable part over  $p$  processors, *perfectly parallelizable*
- Formula of the scaled speedup
- If  $T(n, 1)$  increases monotonically with  $n$  and the sequential part  $c$  is constant, we can achieve linear scaled speedup

- **Scalability**

- Ability to increase the speedup proportionally to the processors number
- Keep efficiency  $E = s/p$  constant when increasing both the processors number and the problem size, such algo are said *scalable*

# Chapter 4

## OpenMP & Cache

- Comparison between matrix multiplication algorithm
  1. Strategy:  $n^2$  do not fit in cache
  2. Strategy:
    - Pro: cache is better
    - Cons:
      - \* Threads are created and destroyed many times
      - \* Race condition, reading and writing the same memory location
      - \*  $n + \#t \cdot n + n$
  3. Strategy:
    - Pro: cache, 1 row and 1 col fit in cache
    - Con: thread, inner loop takes  $n^2$  times overhead due to thread management
    - reduction(+:cij), private so no race condition
  - First strategy best one
  - Second strategy has huge variability since we do not know where the threads will be putted, thus there is the possibility the cache has heavy load already
  - Third strategy worst one
  4. Strategy: Swap the initial two inner loops, reduce the number of creation and elimination of threads
  5. Strategy:
    - num\_thread(num\_thread)
    - nowait, when we finish we move forward the next row
    - barrier, deciding how many rows are processed each time
    - best strategy overall

# Chapter 5

## Pattern of Parallelism I

- Need to design something quite specific for the problem
  - **Decomposition technique**, generate enough sub-problem to be executed in parallel
  - **Mapping**, how is going to execute what maximizing the load balancing

### 5.1 Decomposition

- **Data-Input Decomposition**
- **Data-Output Decomposition**
- **Exploratory**
  - Find the best solution, exploring such space
  - Exploratory decomposition, solution space partitioned and each partition is explored independently
  - Only one task will find the solution the others become useless
    - \* Look for a valid solution, tasks are forced to end
    - \* Look for the best solution, subtree are associated with an expected of the solution, most promising task are explored first
    - \* Task works in parallel, exploration order could be different

#### 5.1.1 Task Dependency Graph

- Model decomposition and they control dependencies
- Not unique for a given problem
  - Node = Task
  - Edges = Task/Data Dependencies
  - Node Labels = Task Computational Cost

### 5.1.2 Parallelism Degree and Critical Path

- **Parallelism Degree:** number of tasks that can be executed in parallel
- *Direct Path in a TDG:* sequence of tasks in the TDG linked by a dependency relation (not in parallel)
- **Critical Path:** longest direct path in the TDG (bottleneck, minimum execution time)
- *Average Parallelism Degree,* work/length of the critical path

### 5.1.3 Task Interaction Graph

- Data Dependencies task exchange data with others in a decomposition
  - Node = Task
  - Edges = Task/Data Dependencies
  - Node Labels = Task Computational Cost
  - Edges Labels = Amount of data exchanged

## 5.2 Mapping

- **TDG:** loading balance and minimizing waiting time
- **TIG:** minimizes interaction/communication between tasks
- **Mapping Guidelines**
  - Independent task to different processors
  - Task in Critical Path assigned as early as possible
  - Minimizing communication cost by scheduling dense subgraphs of the TIG to the same processor
  - **Conflict:** tasks to the same processors reduce the communication cost but does not produce any performance improvement

## 5.3 Prefix-Sum

- Algorithm
- Naive way to parallelize
- Analysis of the speedup
- **Exclusive Prefix Sum**
  - *UpSweep Pass*
    - \* Complexity
    - \* Each node holds the sum of its children

- *DownSweep Pass*
  - \* Every Node should have the sum of all the leaves preceding it (before it left most child)
  - \* Root = 0
  - \* Left child = Parent
  - \* Right child = Parent + UpSweep value of its sibling
- $O(\log N)$  Steps and  $O(N)$  summations
- Large parallelism degree can be exploited

# Chapter 6

## Pattern of Parallelism II - Common Patterns

- **Pipelines**
  - Special kind of task-parallelism
  - The computation is divided into stages that are executed sequentially
  - Asymptotically a pipeline achieves a speedup equal to the number of stages
- **Single Program Multiple Data Embarrassingly Parallel**
  - Problem that can be solved with a large set of complexity independent sub-tasks (data-parallel)
- **Task Pool**
  - *Task List* is stored in a shared data structures
  - Fixed number of threads
  - Each one pick a task and execute it, thus synchronization
  - A thread can generate a new task
- **Dynamic Task Creation**
  - Each task can dynamically create new tasks
  - *Useful for*
    - \* Improve parallelism degree
    - \* Split a long task into smaller ones
    - \* Adapt to non uniform resources
  - A task queue has to be shared among a pool of threads, master-worker perform centralized load balancing
    - \* Remove centralization and favour data exchange among neighbors
    - \* *Push-Sender Initialized* worker that generates a new task sends it to an other worker
    - \* *Pull-Receiver Initialized* when a worker is idle, it asks to other worker for a job to execute (work stealing)



- *Partner Selection*
  - \* Random
  - \* Global Round Robin (GRR)
    - Global variable points to the next worker
    - A worker that needs a partner reads and increments the global variable
    - Implement a GRR
  - \* Local Round Robin (LRR)
    - Every processor keeps a private pointer to the next available worker
    - No overhead due to sharing a global variable
    - Approximate a GRR

## 6.1 Parallelizing Quicksort

### 6.1.1 Sequential Version

- Divide-and-conquer, sort a sequence by recursively dividing it into smaller subsequences
- *Divide*: a sequence is partitioned into two nonempty subsequences such that each element of the first subsequence is smaller than or equal to each element of the second subsequence
- *Conquer*: subsequences are sorted by recursively applying quicksort
- This is accomplished by selecting the pivot and using this element to partition the sequence into two parts - one with elements less than or equal to x and the other with elements greater than the pivot
- **Issues:**
  - Insufficient parallelism degree
  - Load imbalance
    - \* Sub-arrays depends on the pivot selection
    - \* Their length varies and it is difficult to control
    - \* Different length lead to different workloads and imbalance
    - \* Wors case, one task get N elements

### 6.1.2 Parallel Version

- Assign sub-arrays to processors
- Pick a pivot
- *Local arrangement* we divide each sub-arrays in a smaller and a greater part respect the pivot

- *Global arrangement* merge the the left and right part of each sub-arrays in two bigger sub-arrays
- Recursion of the two new sub-arrays until each process can use a sequential algorithm to sort its sub-array
- **Note**
  - Higher parallelism degree
  - Load balance is still sensitive to pivot selection

## 6.2 Odd-Even Transposition

- Based on the bubble sort, which compares every 2 consecutive numbers in the array and swap them if first is greater than the second to get an ascending order array
- Repeat  $n/2$  times:
  - Compare-exchange odd elements with their immediate neighbour
  - Compare-exchange even elements with their immediate neighbour
- Sequential:  $O(n^2)$
- Parallel cost when  $p = n/2$ :  $O(n^2)$
- Complexity of sorting not optimal:  $O(n \log n)$
- Generalized assign batches of  $n/p$  elements to each processor
- Local sort on  $n/p$  elements
- Algorithm across batches with  $p$  phases
- Total cost
- $p = O(\log n)$  parallel is cost optimal  $O(n \log n)$
- **Drawback:** increasing  $p$  need to increase exponentially  $n$  to have scalability

## 6.3 Bitonic Sort

- **Bitonic Sequence**
  - $x_1, \dots, x_j$  is monotonically increasing
  - $x_{j+1}, \dots, x_n$  is monotonically decreasing
- **Bitonic Split**
  - Comparator  $[i : j]$
  - $n/2$  comparator of the kind  $[i : i + n/2]$  for  $1 \leq i \leq n/2$  to a bitonic sequence

- proof
- **Bitonic Merge**
  - Recursive bitonic split until sequence is sorted in increasing order
  - $\log n$  stages, each one perform  $n/2$  compare/swap
  - complexity and parallel time
- **Bitonic Sort for non-Bitonic Sequence**
  - Ascendingly & Descendingly by recursively invoking bitonic sort
  - $\log N$  bitonic merge phases
  - Sequential complexity
  - Parallel Complexity (not optimal)
- Can be parallelized very easily, it exploit a large parallelism degree

# Chapter 7

## Large-Scale Data Processing with Map-Reduce

- Everything is build o top key value pairs
  - $map(k_1, v_1) \rightarrow list(k_2, v_2)$
  - $reduce(k_2, list(v_2), \rightarrow list(k_3, v_3)$
- All in parallel we have a set of mappers and a set of reducers
  - $map(k_1, v_1) \rightarrow list(k_2, v_2)$
  - *shuffle*
  - $reduce(k_2, list(v_2), \rightarrow list(k_3, v_3)$
- **Coordination: master data structures**
  - task status: idle, in-progress, completed
  - Idle get scheduled as workers become available
  - When a map task is completed, it send the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
  - Master ping workers periodically to detect failures
- **Failures**
  - *Map Worker Failures*
    - \* Map task completed or in progress are reset to idle
    - \* Reduce workers are notified when task is rescheduled on another worker
  - *Reduce Worker Failures*
    - \* Only in progress tasks are reset to idle
    - \* A different reducer may take the idle task over
  - *Master Failure*
    - \* Map Reduce task is aborted and client is notified

- **How many Map and Reduce jobs?**

- Make M and R larger than the number of nodes in the cluster
- #Mappers is determined by the #input split
- Many mappers/reducers improve the load balance and speed recovery but increase overhead management
- $R \leq M$ 
  - \* System has a maximum capacity of parallel reducers, executed in waves
  - \* Shuffling of the first wave is done in parallel by mappers
  - \* Shuffling of other waves done later

- **Combiners**

- Map tasks on the same node will produce many pairs with the same key
- We can pre-aggregate information after mapper
- $combine(k_2, list(v_2)) \rightarrow list(k_3, v_3)$

- **Partition Function**

- Inputs are created by continuous split of the input
- For reducer, intermediate(k,v) pairs with the same key end up at the same reducer
- Hash default partitioning function

- **Is Map Reduce so good?**

- Map reduce is not for performance
  - \* Mapper writes to disk, huge overhead
  - \* Shuffle bottleneck
- Little coding time
  - \* Override two functions
  - \* Not everything can be implemented in terms of map reduce
- Fault tolerance

# Chapter 8

## Vertex Centric Paradigm

- Each node knows about its neighbours (graph structure may change)
- Each node may hold some additional custom info
- Pros:
  - Easy to design
  - Large parallelism
  - Can be implemented over map reduce
- Con: local view may lead to sub-optimal results

### 8.1 Label Propagation

- $(X, C_{min})$ , meaning that  $X$  belongs to the connected component with id  $C_{min}$ , initially  $X$  belongs only to itself
- Each node knows its neighbours
- Iterative algorithm
- Algorithm
- $O(d)$  with  $d$  is the diameter

### 8.2 Hash-To-Min

- List of  $(X, C)$ , meaning that  $X$  knows about nodes in the set  $C$ , initially  $X$  knows  $C=X$  plus its neighbours
- Algorithm
- $O(\log d)$  with  $d$  is the diameter

# Part II

## Bruch Part

# Chapter 9

## Ranking

- Ranking Dataset
  - Explicit Feedback: human assessors are presented with a query and a ranked list, and are asked to grade each document with respect to the query, challenge
  - Implicit Feedback: as user interact with a ranking system, collect signals that are indicative of relevance, challenge
- Ranking Metrics
  - **Ranking as a classification or regression**, problem: we do not care on label value, we care only if a document is relevant or not
  - **Rank correlation (kendall's  $\tau$ )**, pairwise classification if the documents are correctly ordered, problem: the bottom and the upper part are equally weighted
  - **Reciprocal rank at K**, position of the first relevant document over the top k position
  - **Precision at rank K**, fraction of documents in the top -k set having  $y_i > 0$
  - **Average Precision at rank K**, gain (tells us if it is a good document) and discount (intuitively it is the probability of been visited)
- Learning to Rank / ML framework
  - Input space
  - Output space
  - Hypothesis space
  - Loss function
- Learning to Rank proprieties
  - Feature based
  - Discriminative training



# Chapter 10

## Ranking Loss Function

- **Point Wise Losses**

1. Input space: feature vector of each single document
2. Output space: relevance of each single document
3. Hypothesis space: feature vector as input and predict the relevance degree (document)
4. Loss function: measure the accurate prediction of the ground truth
  - Ranking as ordinal regression
  - Ranking as binary classification, sigmoid

- **Pair Wise Losses**

1. Input space: pair of documents as feature vectors
2. Output space: pairwise preferences between each doc pair
3. Hypothesis space: bivariate functions, pair of docs as input and output the relative order
4. Loss function: measure the inconsistency between the obtained order and the ground truth
  - Ranking as preference learning: ranknet, ranking svm
  - From ranknet to lambda-rank

- **List Wise Losses**

1. Input space: group of documents associated with the query
2. Output space: relevance degree of all the documents associated to the query, ranked list of all the documents
3. Hypothesis space: Multivariate functions  $h$ , operate on a group of docs and predict their relevance
4. Loss function: can we use gradient descent?
  - Listnet

- **NDCG Consistency**

# Chapter 11

## Representation and Hypothesis Classes

- **Gradient Boosting Decision Trees**
  - Weak learner
  - We can not use gradient descend in a linear function, not differentiable
  - So regression, with loss MSE
- **Neural Network and Pre-Trained Transformer**
  - Representing word as vectors:
    - \* Skip-grap
    - \* Continuous Bag of words
  - Learning word embedding: word2vec
  - *Context Embedding*
    - \* Bert (Bidirectional Encoder Representation for Transformers)
      - Tokenization
      - Input Encoding, wordpiece
      - Model Architecture, deep bidirectional transformer: self-attention and feed-forward
      - Masked Language Model, predict missing word by random masking
      - Next Sentence Prediction: sentence consecutively or not
      - Pre-train and fine tuning
    - \* MonoBert
      - Training identical as the Bert
      - Data fro training and fine tuning are specific tot he target language
    - \* DuoBert
      - Multilingual corpus
      - Language identification
      - Tokenization and input encoding

- Multilingual model architecture
- Multilingual MLM and NSP
- Pretrained and fine tuning

# Chapter 12

## Complexity of Ranking Functions

- Knowledge Distillation: given a large model find a smaller more efficient model that is just as effective
  - Pruning nodes in trees
  - Discarding Redundant Tree in Forest:  $n$  trees, impact score, discard tree with lowest impact score, repeat until reach  $p\%$  is removed
  - Learning a NN from a tree: lambda-mart
- Early Exit Algorithm: perform approximate inference with partial evaluation instead
  - Placing exit points in decision tree: decide to discard a doc at an exit point given a score/rank threshold
  - Placing exit point in layed NN: cascade transformer
- Ranking cascade: apply more complex models to progressively smaller set of documents
  - Less sophisticated ranking functions are cheap but less accurate
  - More sophisticated ranking functions are expensive but more accurate

# Chapter 13

## Retrieval with MIPS: Representation Learning

### 13.1 Sparse Representation

- **Term Matching**
  - TF-IDF
  - BM25
  - TF-IDF and BM25 are inner product of vectors
  - Learning term importance
  - Learning term Frequencies
  - Vocabulary miss match problem
  - **SPLADE (Supervised Progressive Learning for Approximate Dictionary based Entity extraction)**
    - \* Seed terms: start with an initial seed
    - \* Progressive learning: expand the dictionary terms
    - \* Contextual extraction: it consider the contextual info to improve extraction accuracy
    - \* Supervised learning: classifier for entity extraction
    - \* Approximate matching: account for variations in entity names by allowing approximate matching
- **Document Expansion**
  - \* Anticipating queries from text
  - \* **Doc2Query**: automatically generates queries that are representative of the information contained in the doc
    - Pre-trained with LM: bert
    - Fine-tune with D-Q pairs: doc and human generated query, helps the model learn to generate queries that are relevant given docs
    - Masked language model objective: mask word in both doc and query and train the model to predict those
    - Joint D-Q encoding: capture the relationship between the model and the desired query representation

- Generation and ranking: multiple candidates queries and ranks according to the doc

## 13.2 Dense Representation

- Cross-Encoders vs Bi-Encoders
- Representing words as vectors
  - Skip-gram
  - Continuous bag of words
- **Sentence BERT**
  - Pretrained Transformer model: BERT
  - Siamese of triplet network architecture
  - Contrastive objective for fine tuning
  - Sentence Embedding
  - Semantic sentence similarity
- Deep Passage Retrieval
- How to find non relevant document: ANCE

# Chapter 14

## Retrieval with MIPS: Sparse Vectors

- **The Inverted Index**
  - TAAT-Retrieval
  - DAAT-Retrieval
- Dynamic Pruning for TAAT-Early termination
  - Upper Bound
  - Threshold
- Dynamic Pruning for DAAT
  - Assumption: non negativity and zipfian's distribution of items
  - Concepts: upperbound and threshold
  - *Max Score*
  - *WAND*

# Chapter 15

## Retrieval with MIPS: Dense Vectors

- Quantization
  - Distance approximation
  - Query execution
- Product Quantization
  - Subspace quantization
  - Query execution
- IVFPQ
- Clustering methods - Two Step Approximate MIPS
- Graph methods
  - Voronoi Regions
  - Delaunay Graph
  - Approximate the Delaunay Graph