UNIVERSITÀ
DEGLI STUDI
FIRENZE

Mid-term programming assignments

Parallel Programming

General guidelines

- The goal is to understand how to parallelize a program and evaluate the performance of the parallelization
- Assignments can be implemented individually or in pairs (only for 9-credit course)
  - Both team members must contribute equally and understand all aspects of the implementation

Implementation requirements

- Implement both sequential and parallel versions using OpenMP + vectorization techniques
  - Use C/C++ as the primary language
  - Focus on data parallelism, loop optimization, and proper synchronization
  - Apply vectorization where applicable (consider SIMD operations)

Deliverables

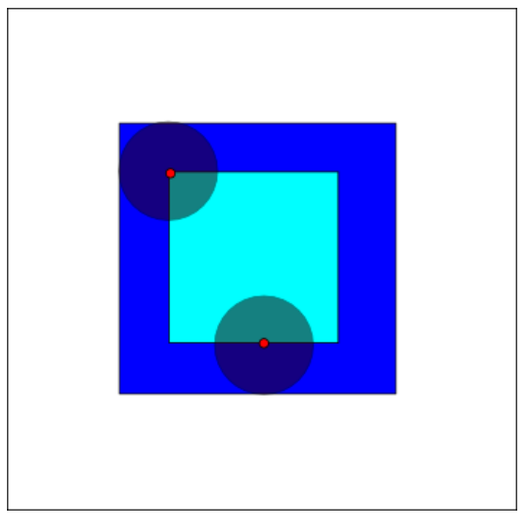
- Source Code: Well-documented, clean code on public repository (GitHub/GitLab/Bitbucket)

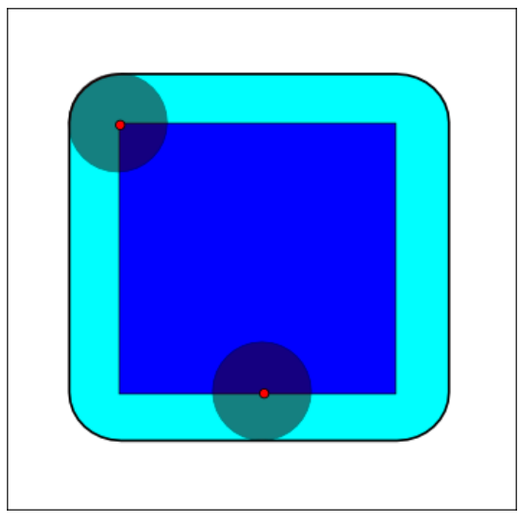
- Technical Report: Comprehensive performance analysis (see detailed guidelines below)

- Presentation: Clear explanation of approach, results, and insights

Morphological image processing

- Implement morphological operations (erosion, dilation, opening, closing) on grayscale or binary images.
  - Implement at least 2-3 morphological operations
  - Test on various image sizes (e.g. 512×512, 1024×1024, 2048×2048, 4096×4096)
  - Consider different structuring element sizes
  - Apply on real images (medical imaging, satellite imagery, etc.)





Morphological image processing

References

- https://en.wikipedia.org/wiki/Mathematical_morphology and https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/GASTERATOS/SOFT/2.htm

- Example applications: https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html

Datasets

- Medical images: https://www.kaggle.com/datasets (search "medical imaging", "X-ray", "MRI")

- Berkeley Segmentation Dataset: https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/

- DIV2K (high-resolution images): https://data.vision.ee.ethz.ch/cvl/DIV2K/

Morphological image processing

Parallelization Opportunities:

- Pixel-level parallelism for operations
- Sliding window parallelism
- Pipeline parallelism for sequential operations

Bigrams / trigrams

- Compute histograms of bigrams/trigrams on texts (eg. Wikipedia / Gutenberg project documents)
  - Consider bigrams/trigrams of characters and words
  - Handle tokenization and normalization (lowercase, punctuation)
- Generate frequency distributions and statistics
  - If needed re-use more and more times the collected data to create a large enough corpus

Bigrams / trigrams

Considerations:

- Token splitting strategies
- Memory-efficient data structures (hash maps, tries)

Links / datasets

- N-gram tutorial: https://web.stanford.edu/~jurafsky/slp3/3.pdf
- Project Gutenberg: https://www.gutenberg.org/
- Wikipedia dumps: https://dumps.wikimedia.org/
- Common Crawl: https://commoncrawl.org/
- Large text corpus: https://wortschatz.uni-leipzig.de/en/download

Bigrams / trigrams

Parallelization Opportunities:

- Document-level parallelism
- Chunk-based text processing
- Parallel hash table updates with atomic operations or locks

ANN retrieval

- Goal: find k-nearest neighbors (k-NN overview: https://scikit-learn.org/stable/modules/neighbors.html) for query vectors in high-dimensional space.
  - Consider high dimensionality vectors (eg. >=128)
  - Support various distance metrics (Euclidean, cosine similarity, Manhattan)
  - Test with k values: 1, 10, 100
- Given a specified query find the *k nearest neighbors* in a database of vectors
  - Can use LSH for approximate indices, eg. using mlpack o LSHkit
  - Can get sample datasets from http://corpus-texmex.irisa.fr

ANN retrieval

Parallelization Opportunities:

- Parallel/vectorized distance computations
- Parallel search across database partitions
- Parallel hash table construction for LSH

Password decryption

- Decrypt passwords encrypted using crypt (DES)
  - Consider 8 characters lengths in the set [a-zA-Z0-9./]. Can focus on specific patterns (dates, common passwords, dictionary words)
  - Library funds may be deprecated or not available. E.g. CUDA does not provide the crypt C-library function. Must use a special implementation (check Moodle).
  - It's a (slowed down) search problem.
  - Can choose to attack a specific password in a list (in different positions), or decrypt a full list of passwords (in this case reduce the search space)
  - Assume salt is known.

Password decryption

Datasets:

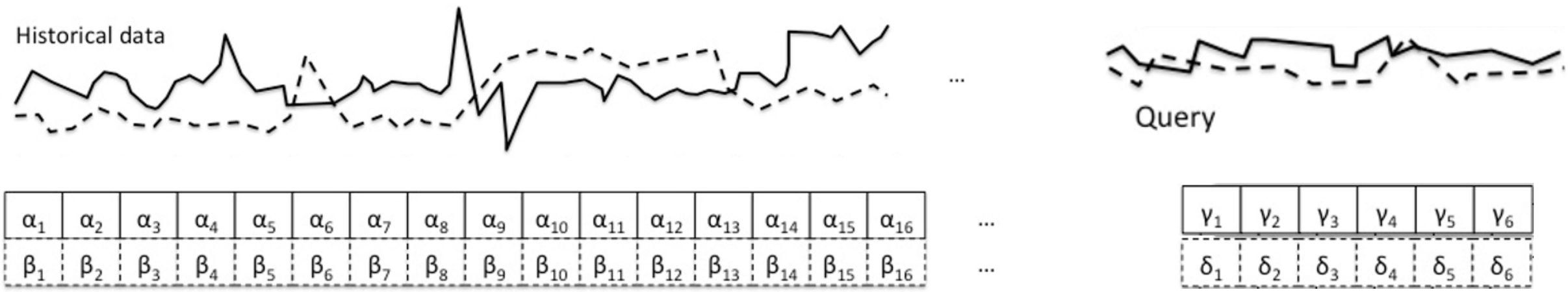
- Use common password lists (e.g., RockYou dataset samples)
- Generate encrypted passwords for testing
- Focus on realistic password patterns to reduce search space

Parallelization Opportunities:

- Search space partitioning
- Task parallelism for multiple passwords
- Load balancing strategies for irregular workloads

Pattern recognition in time series

- Search a given time series within a larger and longer set of time series
  - Implement sliding window search
  - Use SAD (Sum of Absolute Differences) or similar metrics (DTW optional)



Pattern recognition in time series

- Handle multiple pattern queries
- Work with time series of length 1.000-100.000 points

Datasets:

- UCR Time Series Archive
- Stock market data
- Sensor data (IoT, weather)
- Generate synthetic data: https://github.com/cyrilou242/mockseries

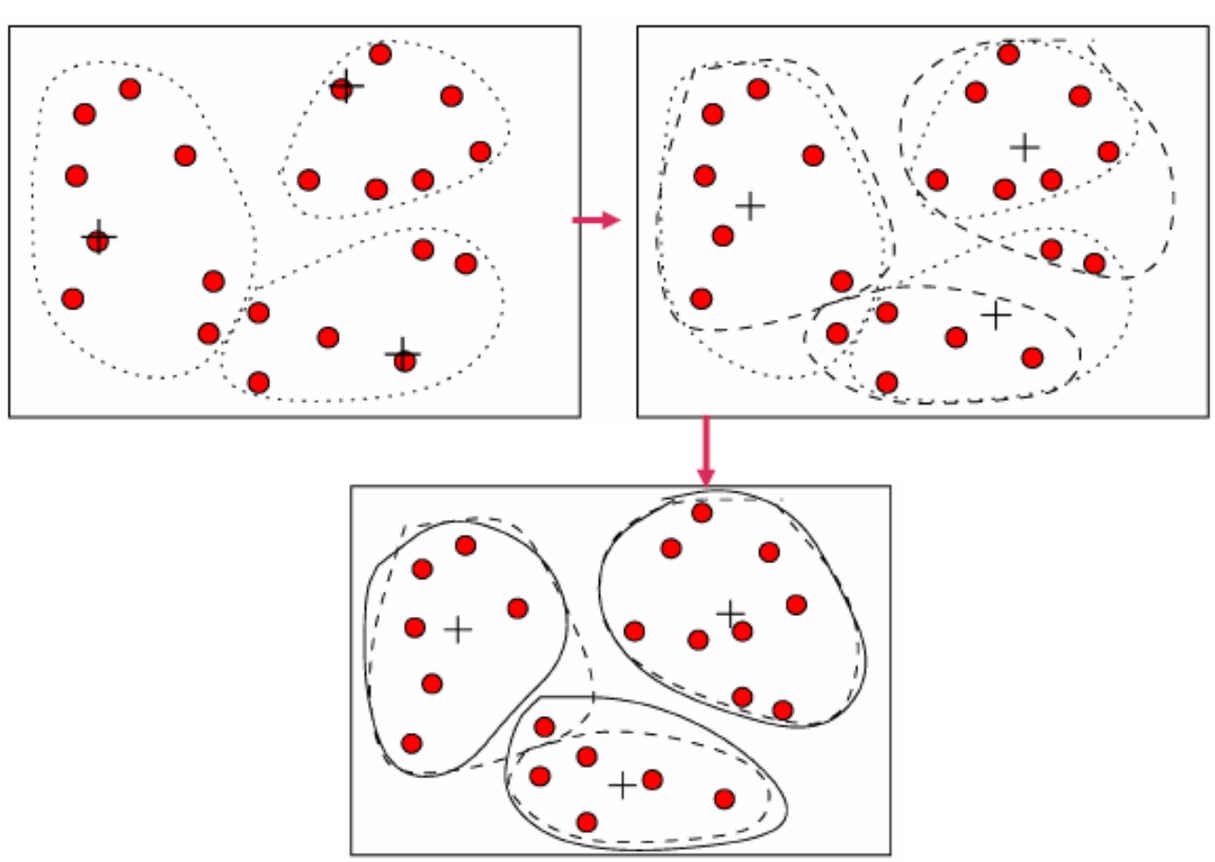
Pattern recognition in time series

Parallelization Opportunities:

- Parallel sliding window evaluation
- Multiple pattern search in parallel
- Data parallelism across time series database

k-means

- Implement a parallel version of k-means clustering
  - There’s no need to implement K-means++ centroid assignments
    - Careful with experiment reproducibility: use fixed iterations or the same starting points
  - Feel free to chose to operate on 2D or larger dimensionality vectors



k-means

Datasets:

- Iris, Wine, Digits datasets
- Image segmentation (pixel clustering)
- Synthetic blob datasets, e.g. https://python-course.eu/machine-learning/artificial-datasets-with-scikit-learn.php

Parallelization Opportunities:

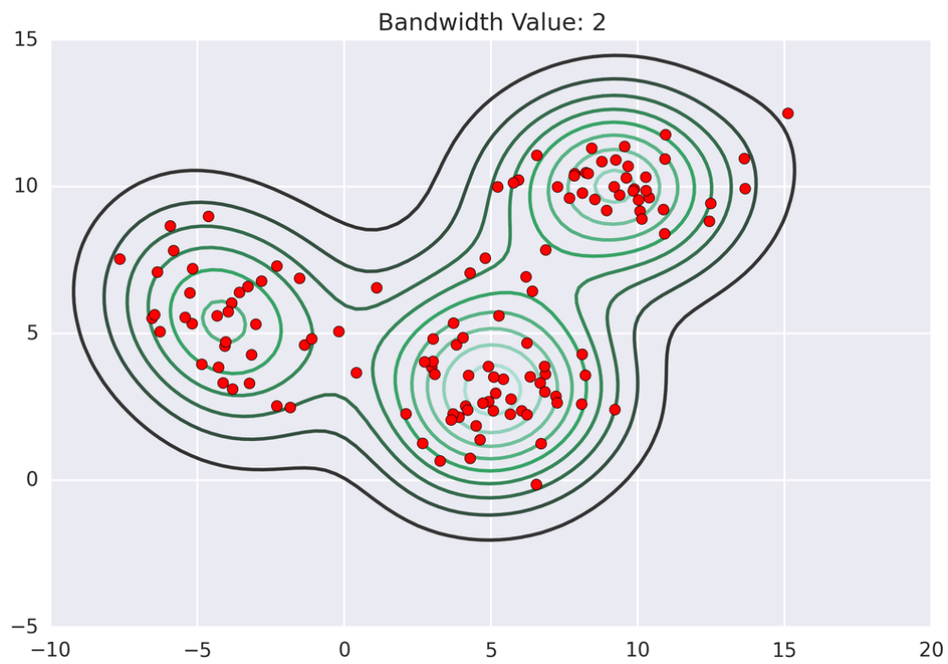
- Parallel distance computation for assignment step
- Parallel centroid update
- Reduction operations for cluster statistics
- AoS vs SoA

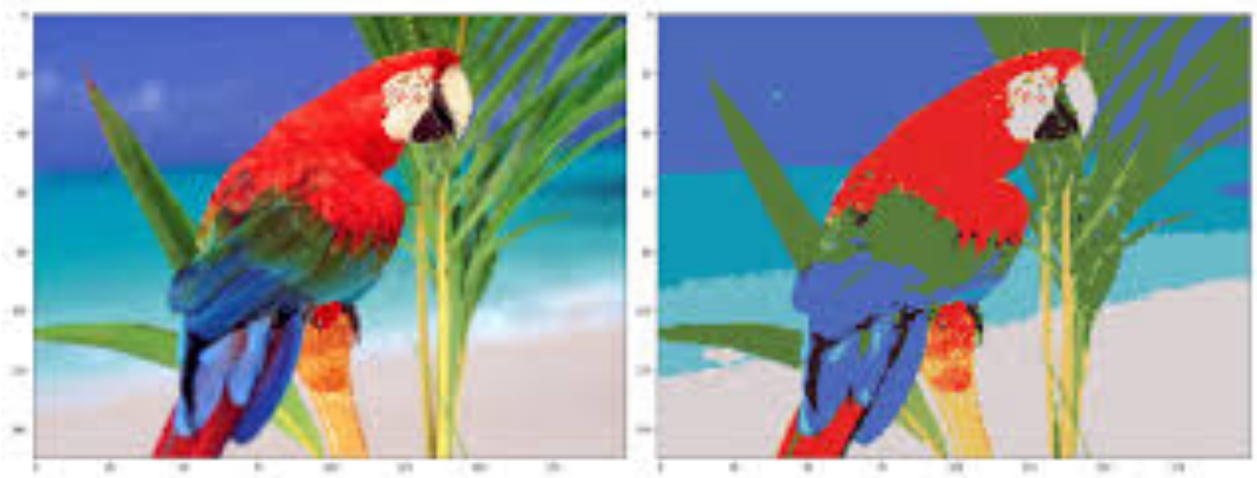
Mean shift clustering

- Implement a parallel version of mean-shift clustering.
  This clustering doesn't require to specify the number of desired clusters (it uses KDE, so needs a bandwidth parameter) but its complexity is $O(N^2)$

  - Test various bandwidth values

- Can be used to segment images



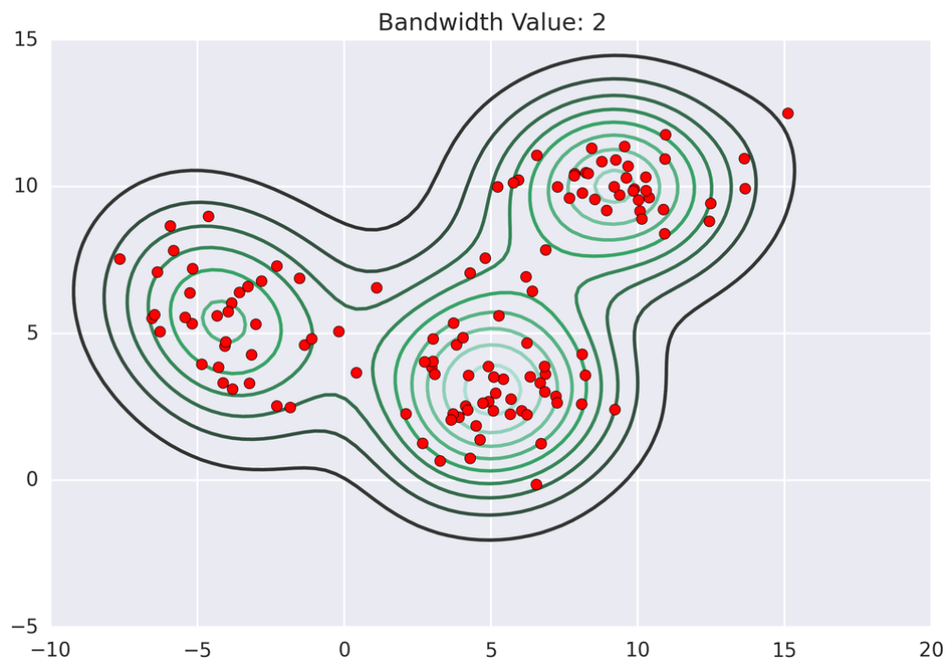


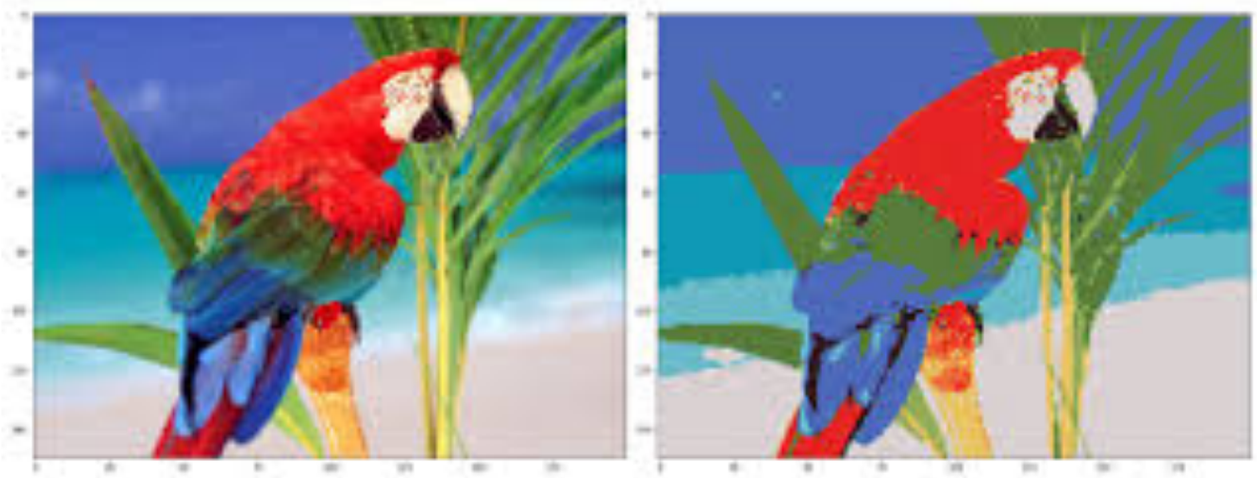
Mean shift clustering

- Implement a parallel version of mean-shift clustering.
  This clustering doesn't require to specify the number of desired clusters (it uses KDE, so needs a bandwidth parameter) but its complexity is $O(N^2)$

  - Test various bandwidth values

- Can be used to segment images





Mean shift clustering

Datasets:

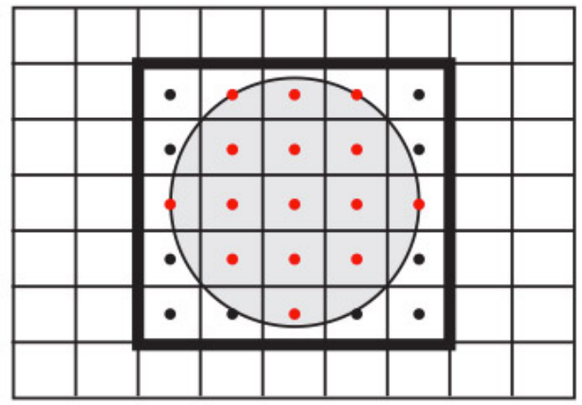
- Berkeley Segmentation Dataset
- COCO dataset samples
- Synthetic 2D/3D point clouds

Parallelization Opportunities:

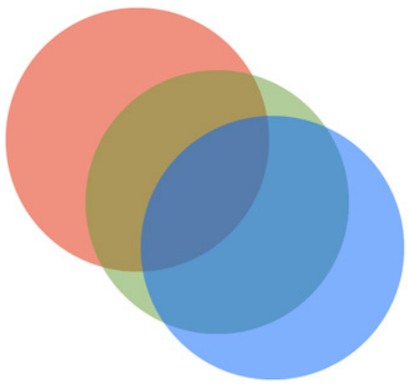
- Parallel mean computation for each point
- Parallel kernel density estimation
- Mode detection parallelism
-

Renderer

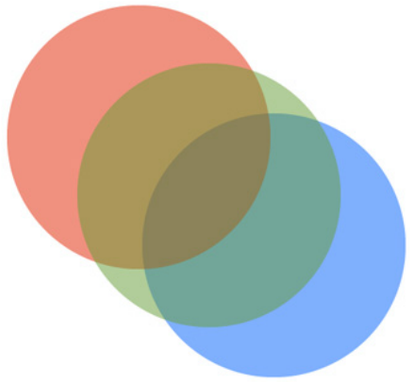
- Implement a renderer that draws (semi) transparent circles (or other shapes). Circles have 3D coordinates and the order along Z axis matters for the correct rendering



- A pixel belongs to a circle if its center is within the circle.



Correct order:
blue over green over red



Incorrect order

Random 10K

Random 100K

Renderer

- Handle transparency/alpha blending
- Sort by Z-axis for correct rendering
- A pixel belongs to a shape if its center is within the shape

Output:

- Generate images at various resolutions (e.g. 512×512, 1024×1024, 2048×2048)
- Support hundreds to thousands of shapes

Parallelization Opportunities:

- Pixel-level parallelism
- Shape-level parallelism with synchronization
- Tile-based rendering

Image composition

- Goal: parallel alpha composition of multiple images (variation of Renderer assignment).

- Implement a parallel alpha blending, supporting multiple overlay images

- Useful to create image augmentation to train Convolutional Neural Networks, e.g. for detection (see Blend augmentation in ImgAug library)

- Can parallelize applying the same object to different images, or in different positions in the same image

Image composition

Datasets:

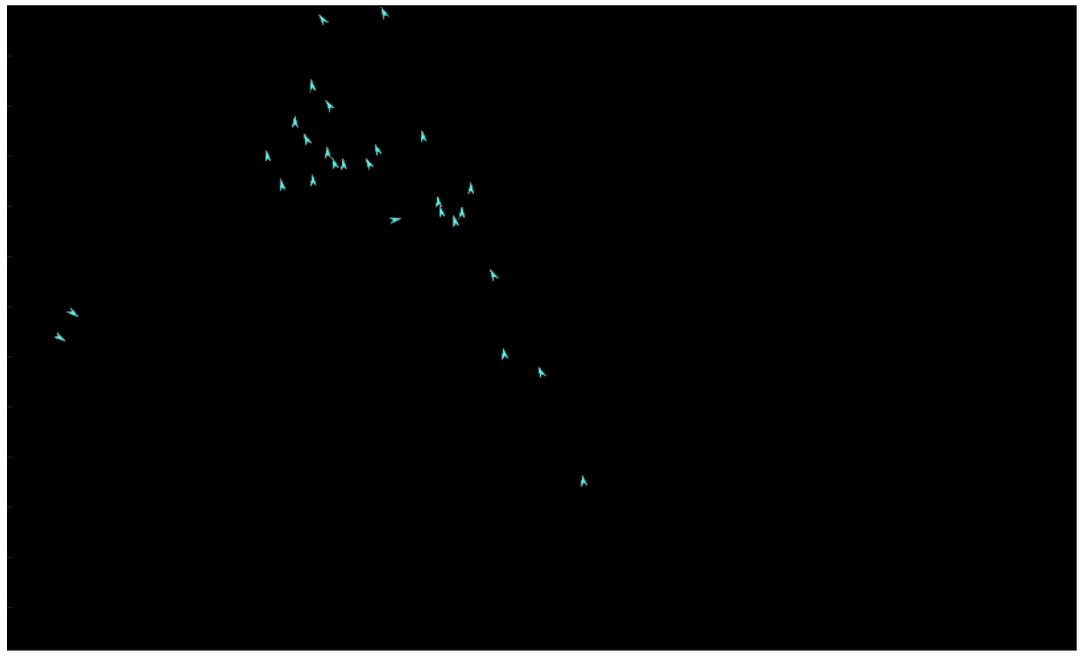
- COCO dataset
- Pascal VOC
- Custom object cutouts with alpha channels

Parallelization Opportunities:

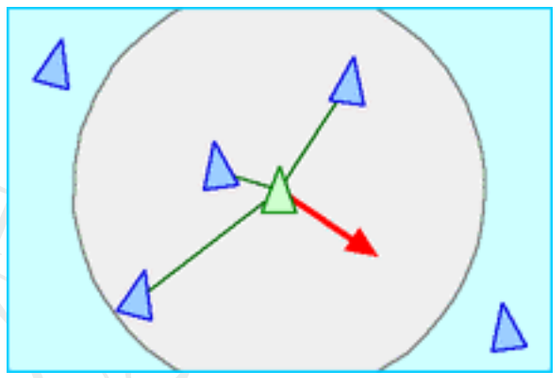
- Parallel composition of image pairs
- Batch processing of augmentations
- Pixel-level parallelism

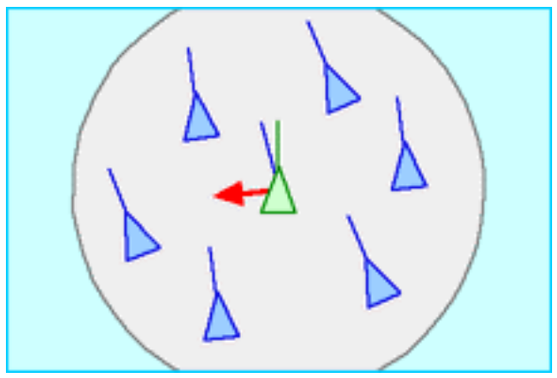
Boids

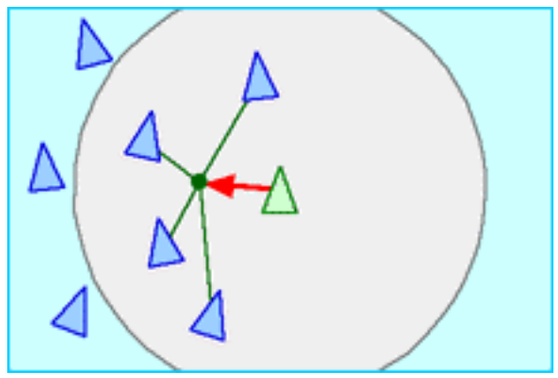
- Simulate the behavior of a flock / crowd.
  - Boids is an artificial life program which simulates the flocking behaviour of birds, and related group motion.



- Check https://vanhunteradams.com/Pico/Animal_Movement/Boids-algorithm.html

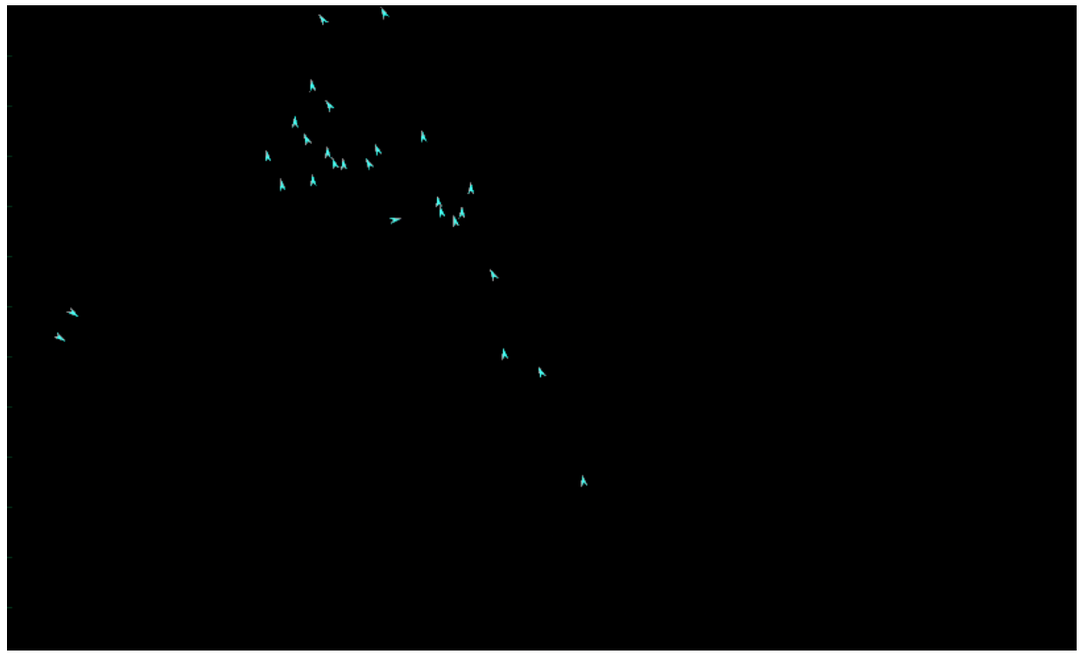




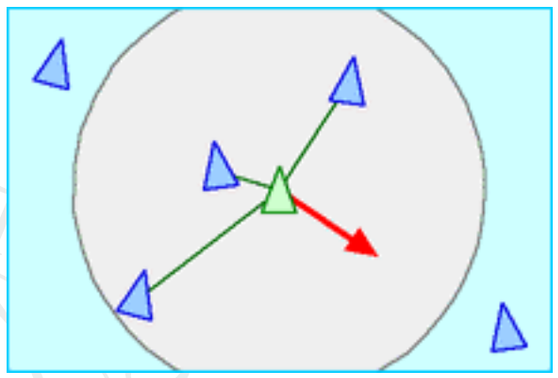


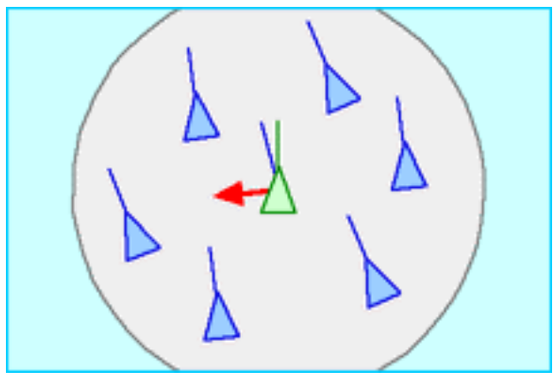
Boids

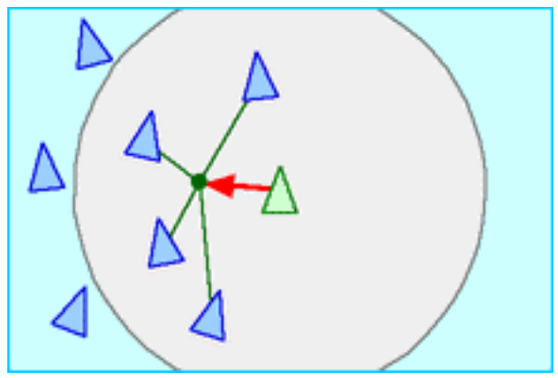
- Simulate the behavior of a flock / crowd.
  - Boids is an artificial life program which simulates the flocking behaviour of birds, and related group motion.



- Check https://vanhunteradams.com/Pico/Animal_Movement/Boids-algorithm.html







Boids

- Implement three basic rules: separation, alignment, cohesion
- Simulate 1000-10000+ agents
- Run for 1000+ time steps
- Optional: visualization of simulation

Parallelization Opportunities:

- Parallel neighbor finding (spatial partitioning)
- Parallel rule computation per agent
- Parallel position/velocity updates

Non-Maximum Suppression (NMS) for Object Detection

- Goal: Implement parallel NMS to filter overlapping bounding box predictions in object detection pipelines.
- Applications:
  - Post-processing in object detection (YOLO, Faster R-CNN, SSD)
    - Any detection pipeline producing multiple overlapping predictions
  - Critical component in many modern object detectors
    - Recent methods are NMS-free
- Tutorial: https://learnopencv.com/non-maximum-suppression-theory-and-implementation-in-pytorch/

Non-Maximum Suppression (NMS) for Object Detection

- Challenges:
  - Sequential nature of greedy NMS (each iteration depends on previous)
  - Race conditions when multiple threads try to suppress same box
  - Load balancing (number of boxes per class/region varies)
  - Memory access patterns (scattered box data)
- Requirements:
  - Standard NMS (greedy sequential algorithm as baseline)
  - Implement parallel variants:
    - Parallel NMS with atomic operations
    - Soft-NMS (weighted score reduction instead of hard suppression)
    - Class-agnostic and class-specific NMS
- Process detection outputs with varying numbers of boxes:
  - 100, 1000, 10,000, 100,000 candidate boxes
- Support multiple IoU (Intersection over Union) thresholds: 0.3, 0.5, 0.7
- Handle multi-class scenarios (80+ classes like COCO)

Non-Maximum Suppression (NMS) for Object Detection

- Parallel NMS Strategies:
  - Optimistic Parallel NMS:
    - Process boxes in parallel
    - Use atomic operations to mark suppressed boxes
    - May miss some suppressions but faster
  - Spatial Partitioning:
    - Divide image into grid cells
    - Apply NMS independently in each cell
    - Handle boundary cases carefully
- Soft-NMS:
  - Instead of removing boxes, decay their scores
  - All boxes processed independently
  - More amenable to parallelization
- Sorted List Approach:
  - Sort all boxes by confidence
  - Process in chunks, each chunk in parallel
  - Synchronize between chunks

Non-Maximum Suppression (NMS) for Object Detection

Datasets:

- COCO detection results (download pre-computed detections)
- Pascal VOC detection outputs
- Generate synthetic detection boxes with known ground truth
- Real detector outputs from pre-trained models (YOLO, Faster R-CNN)

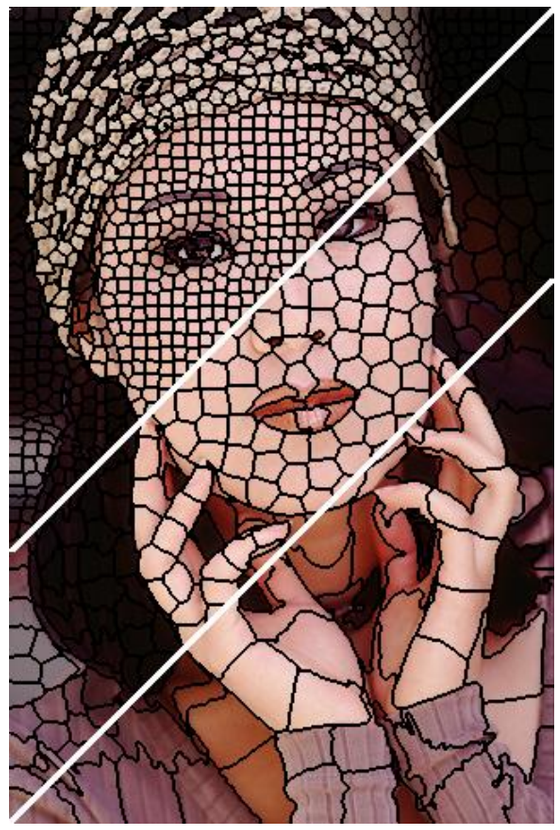
Good Practices:

- Validate correctness against sequential NMS (should get identical or very similar results)
- Measure precision/recall impact of parallel approximations
- Profile IoU computation (often the bottleneck)

SLIC superpixel

- Implement the SLIC algorithm (k-means in CIE-Lab+XY) to segment images into K superpixels.

- SLIC (Simple Linear Iterative Clustering) reduces an image to a few hundred/thousand superpixels - i.e. compact, edge-aware regions useful in many CV pipelines. The algorithm is iterative, data-parallel, and memory-bandwidth bound



- SLIC superpixel tech report: https://infoscience.epfl.ch/entities/publication/2dd26d47-3d00-43eb-9e31-4610db94a26e

- Comparison of segmentation and superpixel algorithms: https://scikit-image.org/docs/0.25.x/auto_examples/segmentation/plot_segmentations.html

SLIC superpixel

- Datasets:
  - Berkeley Segmentation (BSDS500 - https://www.kaggle.com/datasets/balraj98/berkeley-segmentation-dataset-500-bsds500),
    COCO samples, other high-res photos.

Parallelization Opportunities:

- Parallel assignment step over pixels.
- Parallel cluster update using reductions.
- Experiment with tile/block processing to improve cache locality.
- SoA vs. AoS: how to apply it to an image ?

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Implementation guidelines

Dataset

Finding Appropriate Datasets:

- Size matters: Ensure dataset is large enough to show meaningful parallelization benefits
- Rule of thumb: computation time should be ≥10 seconds sequential for good measurements
- If dataset is too small, replicate or synthesize larger versions

Realism: Use real-world datasets when possible for practical relevance

- UCI Machine Learning Repository
- Kaggle datasets
- OpenML datasets
- Government open data portals
- Synthetic data: When real data is unavailable, generate synthetic datasets

Data Preprocessing:

- Document all preprocessing steps
- Ensure preprocessing doesn't dominate computation time
- Consider whether preprocessing should be parallelized separately

Parallelization Parameters

Number of Threads:

- Test: 1, 2, 4, 8, 16, 32 threads (up to 2× physical cores)
- Identify optimal thread count
- Observe superlinear speedup (if any) or speedup saturation

Scheduling Strategies (OpenMP):

- schedule(static): Equal-sized chunks, predetermined
- schedule(dynamic): Dynamic load balancing
- schedule(guided): Decreasing chunk sizes
- schedule(static, chunk_size): Experiment with chunk sizes
- Test which works best for your workload distribution

Chunk Sizes (when using static or dynamic scheduling):

- Too small: excessive overhead
- Too large: poor load balancing
- Test: automatic, manual specification (powers of 2, problem-dependent)

Parallelization Parameters

Memory Access Patterns:

- Compare row-major vs column-major access
- Test effect of data structure layout (AoS vs SoA)
- EXTRA: Analyze cache hit rates

Synchronization Strategies:

- Critical sections vs atomic operations vs reduction clauses
- False sharing mitigation (padding, alignment)
- Lock granularity (fine-grained vs coarse-grained)

Vectorization Parameters:

- Test with/without vectorization flags
- EXTRA: Analyze vectorization reports

Performance metrics

Execution Time:

- Wall-clock time (most important for users)
- CPU time (sum of all thread times)
- Measure multiple runs (minimum 5) and report statistics (mean, std, min, max)

Speedup:

- Speedup(p) = T_sequential / T_parallel(p)
- Report speedup curves (speedup vs thread count)

Scalability

- Strong scaling: fixed problem size, increasing threads
- EXTRA: Weak scaling: problem size grows proportionally with threads

Performance metrics

Timing

- Use high-resolution timers: `omp_get_wtime()`, `std::chrono::high_resolution_clock`
- Warm-up runs: discard first 1-2 runs to eliminate cold-start effects
- Multiple measurements: run at least 5-10 times, report median or mean with confidence intervals
- Measure only computation, exclude I/O and initialization when possible

System Configuration

- Document hardware: CPU model, core count, cache sizes, RAM, compiler version
- Close unnecessary background processes
- If possible disable CPU frequency scaling

Technical report

Abstract

- Brief problem description
- Approach summary
- Key results (best speedup achieved)

Introduction

- Problem description and motivation
- Computational challenges
- Objectives and scope

Algorithm Description

- Sequential algorithm explanation
- Identification of parallelizable sections

Parallel Implementation

- Parallelization strategy
- OpenMP directives used (#pragma omp parallel for, reduction, etc.)
- Synchronization mechanisms
- Vectorization techniques applied
- Data structures and memory layout choices
- Challenges encountered and solutions

Experimental Setup

- Hardware specifications
- Software environment (OS, compiler, flags)
- Datasets used (description, sizes, sources)
- Parameters tested
- Measurement methodology

Technical report

Results and Analysis

- Correctness validation: how you verified correctness
- Performance results:
- Speedup curves (speedup vs threads) with multiple input sizes
- Scalability analysis (strong/weak scaling)
- Effect of different parameters (scheduling, chunk size, etc.)
- Comparison of variants: different parallelization strategies
- Bottleneck analysis: what limits scalability?
- All plots should have clear labels, legends, and captions

Discussion

- Interpretation of results
- Comparison with theoretical expectations (Amdahl's Law, Gustafson's Law)
- Overhead analysis
- Limitations and constraints
- Lessons learned
- Conclusions
- Summary of achievements
- Best configuration found
- Future work suggestions

References

- Academic papers, documentation, dataset sources

Presentation

- Duration: 10-12 min + 3-5 min. Questions/ discussion

Structure:

- Problem Introduction (1-2 min)
    - What problem are you solving?
    - Why is it computationally challenging?
- Approach (3-4 min)
    - Sequential algorithm overview
    - Parallelization strategy
    - Key optimizations
- Results (4-5 min)
    - Speedup curves (main result)
    - Best configuration achieved
    - Interesting findings or surprises
- Conclusions (1-2 min)
    - Summary
    - Lessons learned

Tips:

- Visualizations: Use clear graphs, avoid cluttered plots
- Focus: Don't explain every detail, highlight key insights. Probably you do not need to copy/paste code !
- Demo: If possible, show a quick live demo or image/video
- Rehearse: Practice timing
- Keep extra slides for discussion/answering details

Code

- Version control: small, descriptive commits; tags for milestones.
- Documentation: short README, build/run instructions, parameter table, dataset citations.
- OpenMP official documentation: https://www.openmp.org/specifications/
- OpenMP tutorials: https://computing.llnl.gov/tutorials/openMP/

Common pitfalls to avoid

- Measurement Errors:
  - Not running enough iterations for stable measurements
  - Including I/O time in performance measurements
  - Not accounting for cold cache effects
  - Comparing debug vs release builds
- Implementation Issues:
  - Race conditions due to missing synchronization
  - False sharing causing performance degradation
  - Incorrect use of private vs shared variables in OpenMP
  - Not validating correctness of parallel implementation
- Analysis Problems:
  - Testing only on small inputs that don't show parallelization benefits
  - Not exploring different thread counts systematically
  - Ignoring load imbalance issues
  - Not comparing different scheduling strategies
- Report Issues:
  - Missing hardware specifications
  - Plots without labels, legends, or captions
  - No discussion of why certain results occurred
  - Not explaining deviations from expected behavior

Evaluation criteria

- Correctness (~25%)
  - Correct sequential implementation
  - Correct parallel implementation (validated against sequential)
  - Proper handling of edge cases
- Performance (~25%)
  - Achieved speedup
  - Efficiency of parallelization
  - Appropriate use of OpenMP features
  - Evidence of optimization effort
- Experimental Methodology (~20%)
  - Comprehensive parameter exploration
  - Proper measurement techniques
  - Statistical rigor
  - Variety of test cases
- Report/Presentation Quality (~20%)
  - Clear writing, organization and presentation
  - Thorough analysis and discussion
  - Quality of visualizations
  - Completeness
- Code Quality (~10%)
  - Readability and documentation
  - Repository organization