Non Maximum Suppression

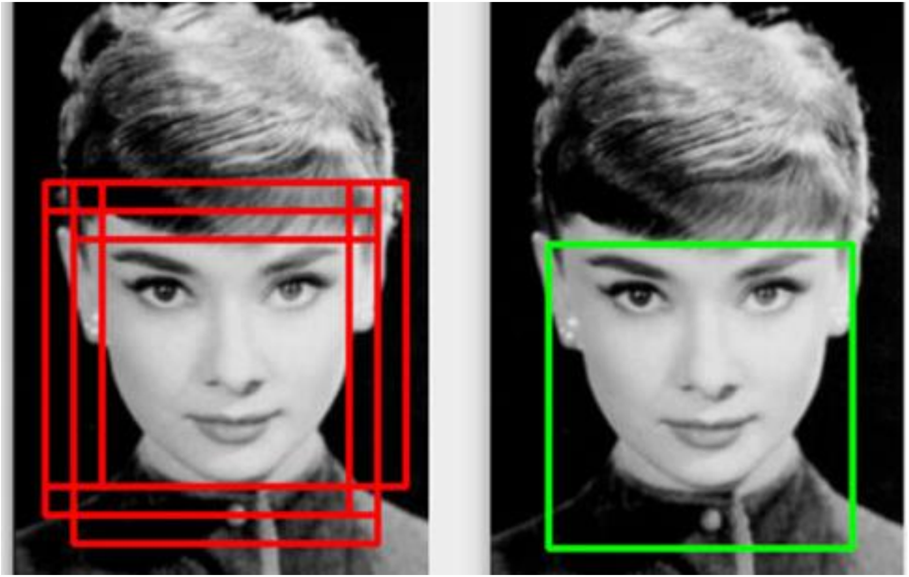
Seth Park
Vy-An Phan
Tushar Singal
Lichang Xu
Jerry Zhao
Simon Zimmerman

Advisor: Bichen Wu

Overview

- **Project Description**
- Algorithm
- Serial Code
- Parallelization
- Results

Project Description: What is NMS?



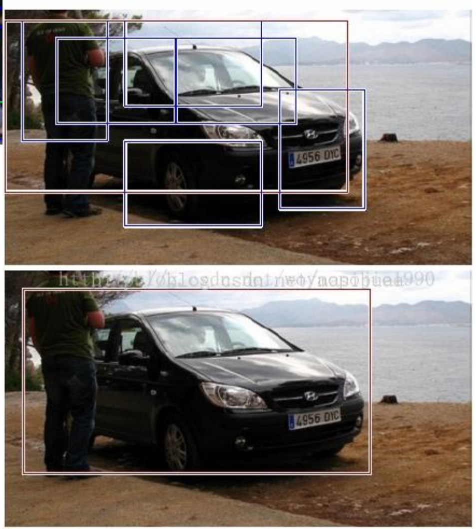
- Non-Maximum Suppression
- An intermediate step in many edge detection algorithms
- Object detection may result in multiple results (shown by red bounding boxes)
- NMS reduces them down to one box

Project Description: Applications

- NMS can get computation heavy!
- Our application: detection of obstacles from a self-driving car’s camera
- Requires speed for real-time data analysis and quick reactions

Project Description: Other applications

- Facial recognition
- Feature extraction
- Satellite data analysis
- Depth analysis
- Medical scans



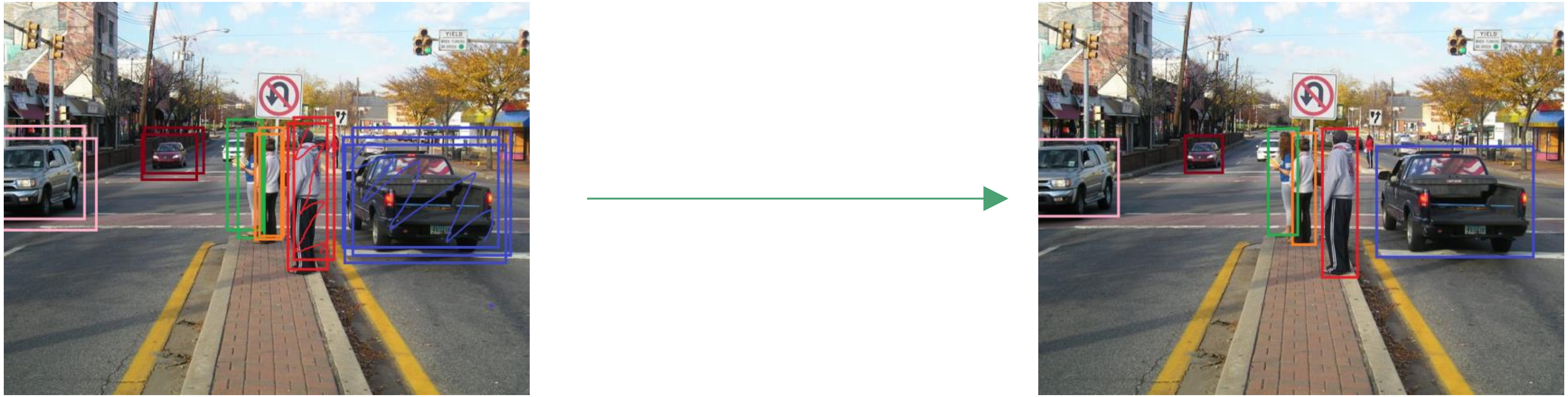
Overview

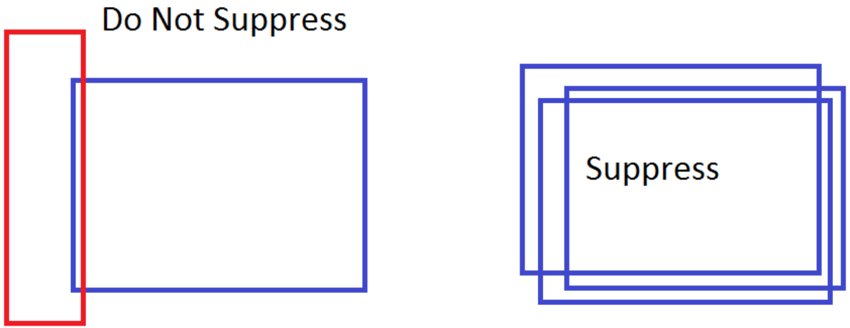
- Project Description
- **Algorithm**
- Serial Code
- Parallelization
- Results

Algorithm: NMS

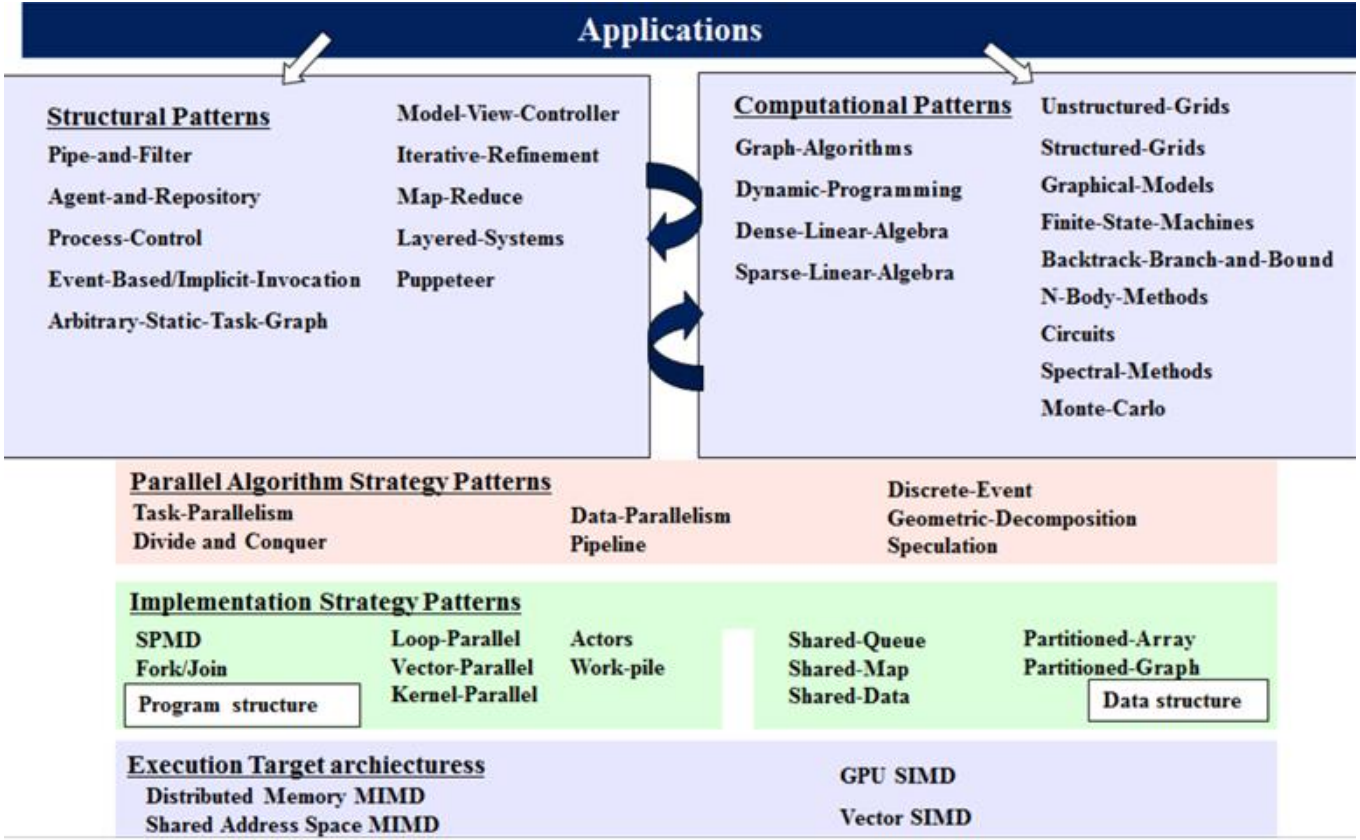
- Takes a list of bounding boxes (detected elsewhere)
- Loop over bounding boxes and compute overlap ratios
    - Overlap ratio threshold is determined by the user
- Boxes that overlap too much (overlap > threshold) are suppressed
- Boxes detected with a higher probability are favored
- Return a new list ignoring the redundant bounding boxes

Algorithm: NMS

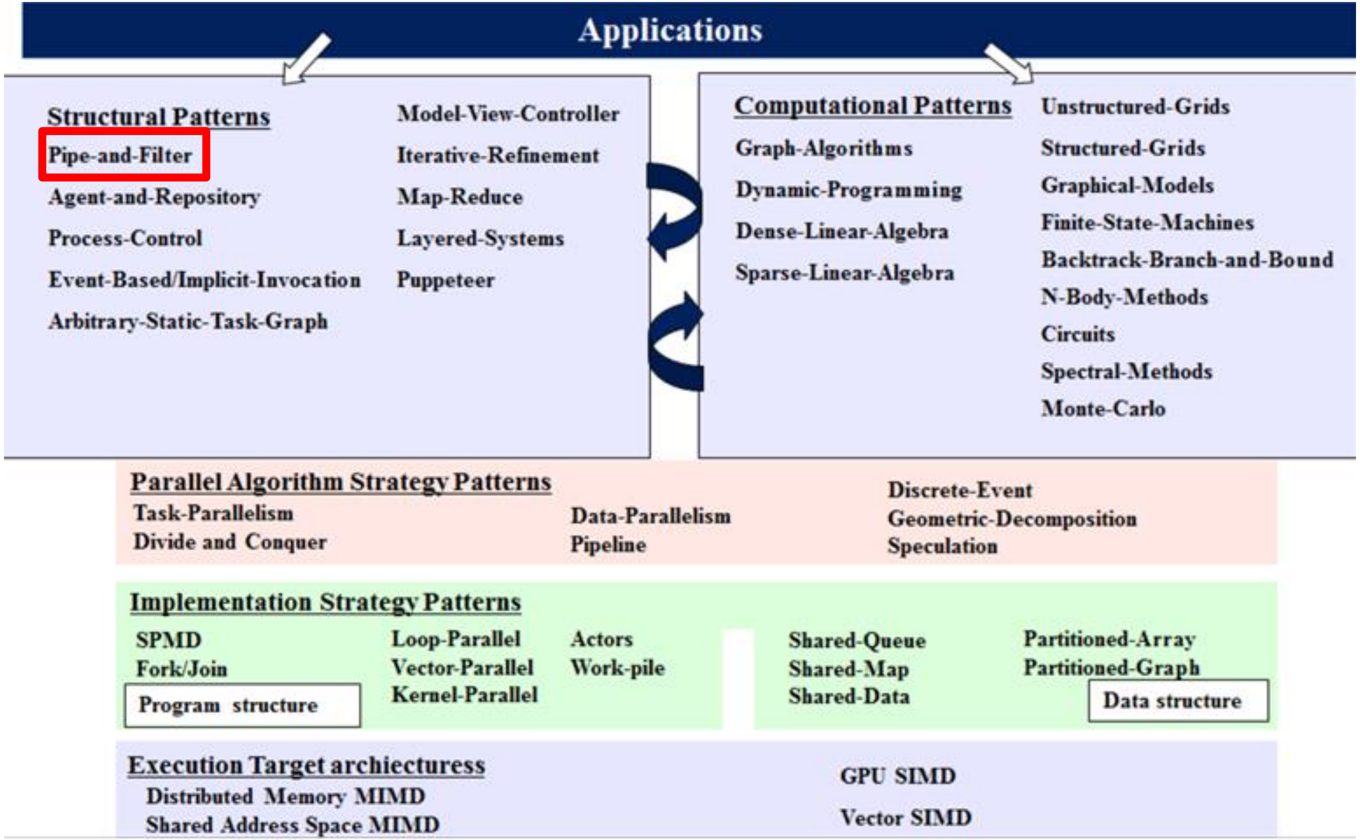




Algorithm: Patterns



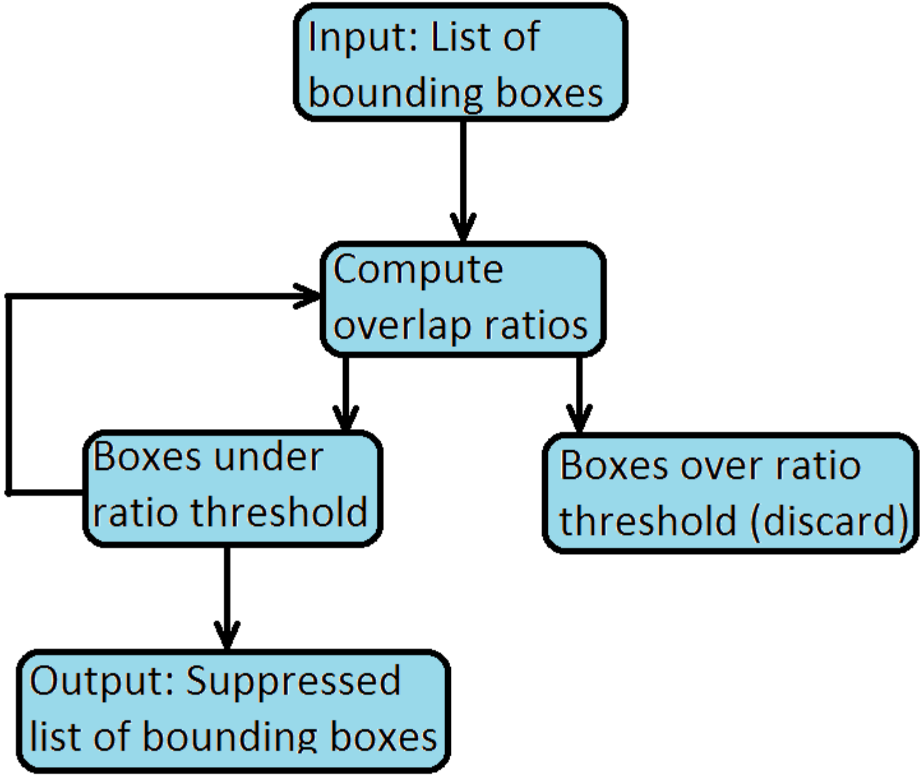
Algorithm: Patterns



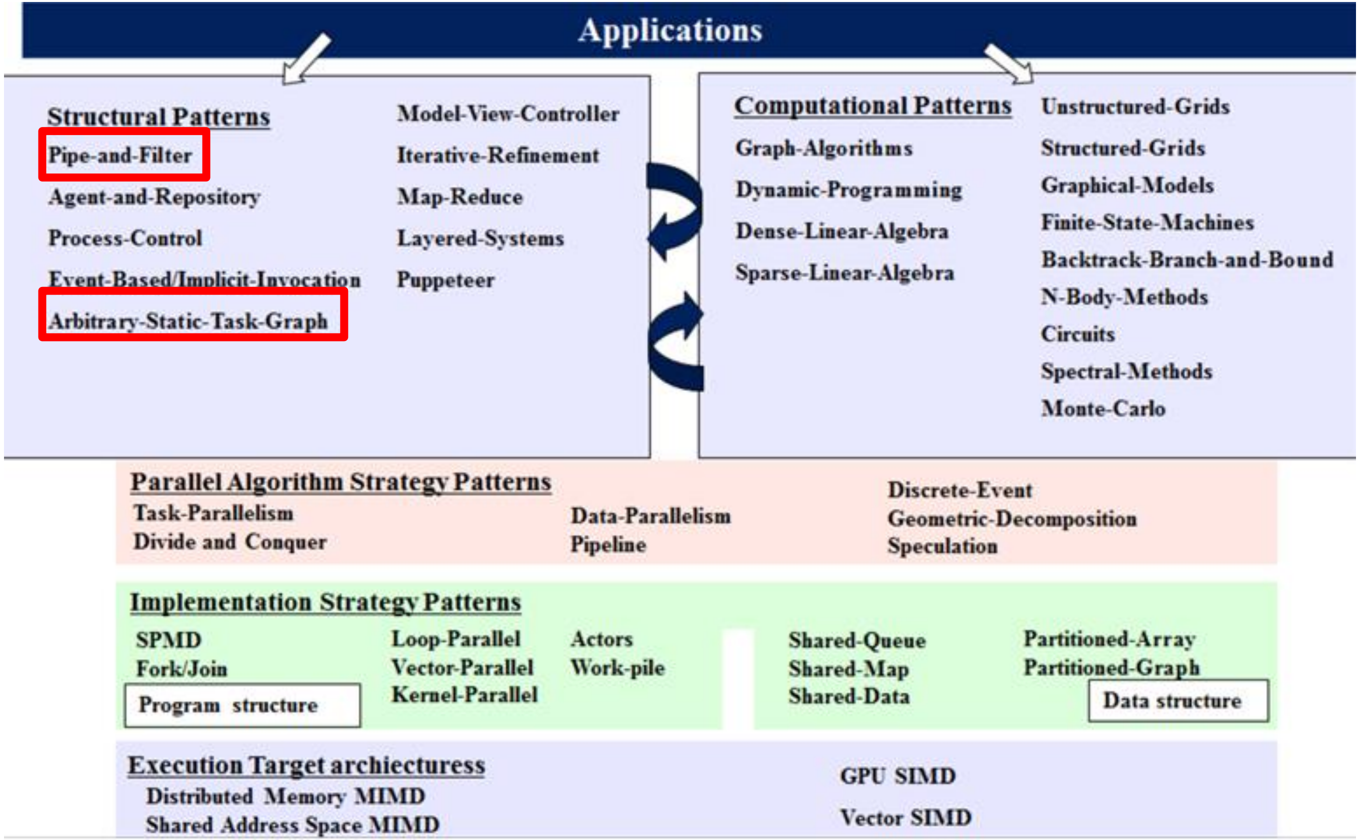
Algorithm: Patterns and Strategy

Despite the iterative step of this algorithm, it cannot be classified as iterative refinement because all comparisons must be performed.

There is no "refinement level" to reach

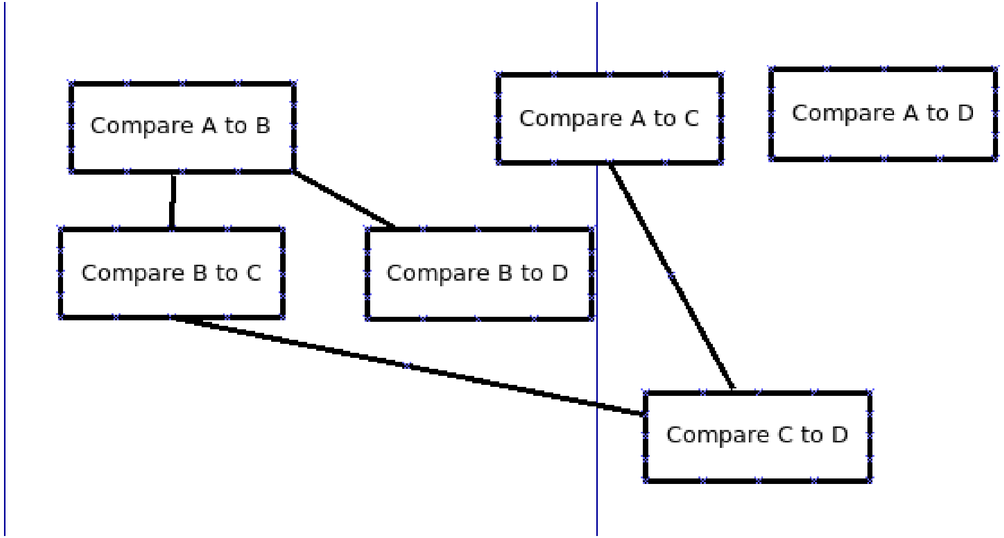


Algorithm: Patterns

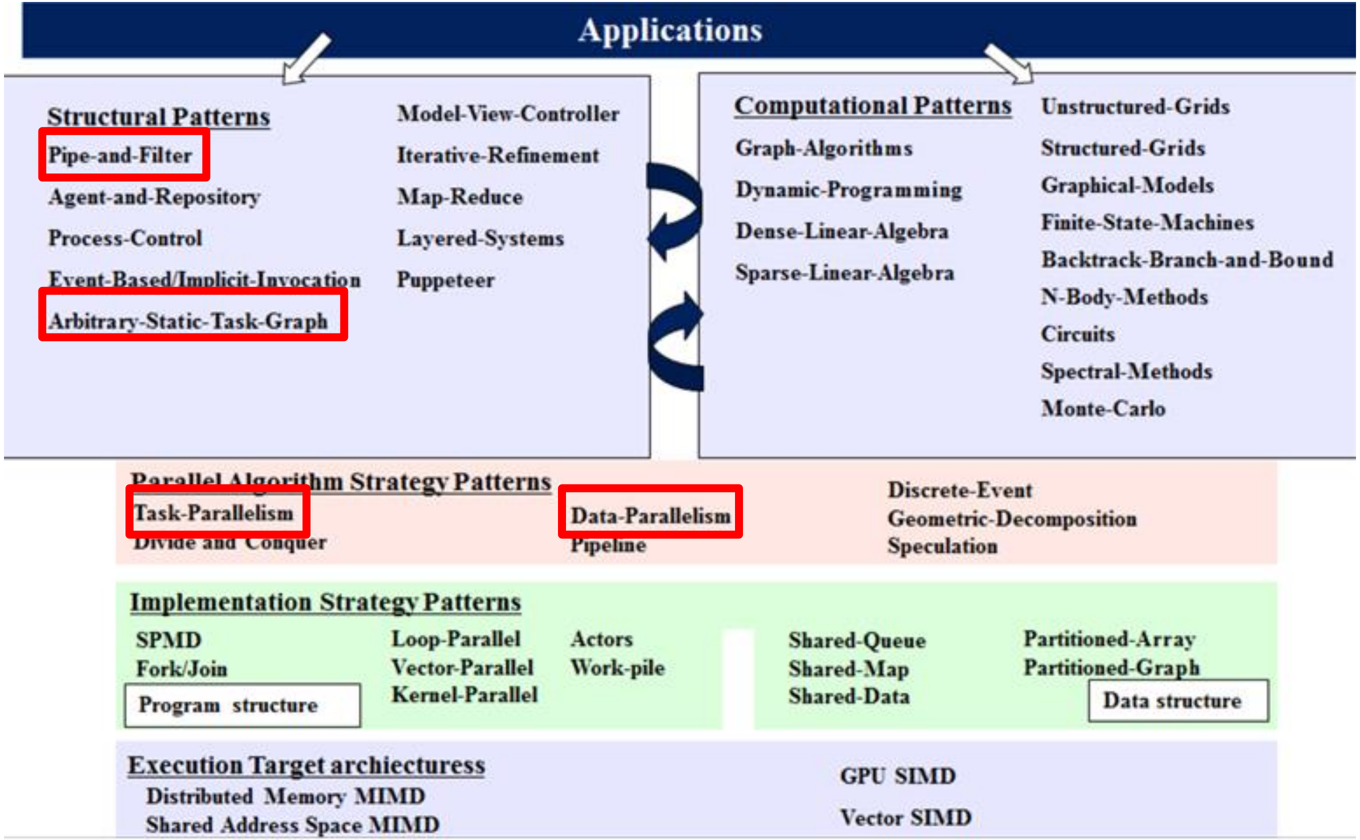


Algorithm: Patterns

- Each comparison is its own task
- Future comparisons dependent on results of previous comparisons
- Cut away tasks/comparisons as you realize they are unnecessary



Algorithm: Patterns



Algorithm: Patterns

- Task parallel - two possibilities
  - Assign tasks to comparisons; each comparison is a task
  - Assign tasks to boxes; determining if we want to keep box X is a task
- Data parallel
  - We compare box i to all other boxes
  - Box i is represented by 4 arrays, xmins[i], ymins[i], widths[i], heights[i]
  - Each comparison is data parallel over the dimensions of the other boxes

Overview

- Project Description
- Algorithm
- **Serial Code**
- Parallelization
- Results

Serial Code: Initial Layout

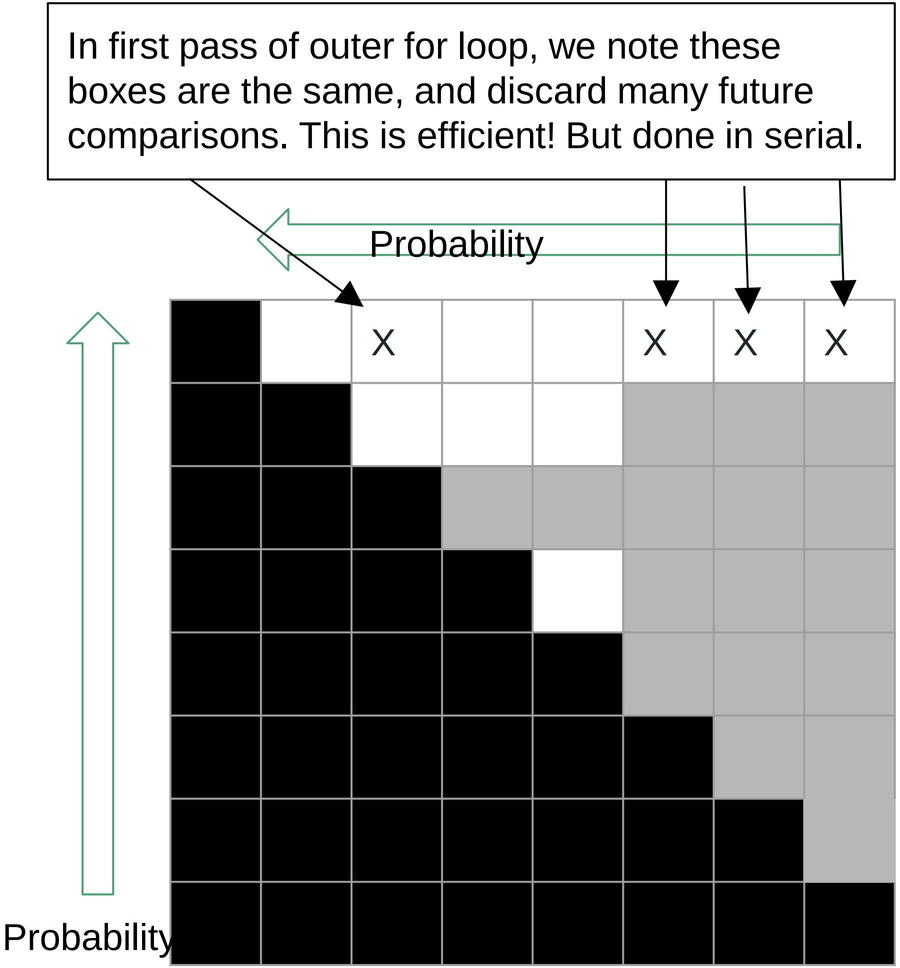
- We started with python serial code
- Double-nested for loop and lots of memory accesses
- Took about 5min to run for 100 images, each with ~15000 boxes
    - ~3s per image
- This method cannot react in real-time for a car camera going at 60 fps
- Translate to C
    - Simple translation, same logic
    - No parallelism, but got ~100x speedup anyway simply by language choice
    - Compiled with -O2 -funroll-loops

Serial Code: Diagram

Initially, we believe we should keep every box. The boxes are ordered by their detection probabilities. (Boxes[0] has highest probability)

```
for i if keep[i] == True:
    for j > i if keep[j] == True:
        if R > iou_compare(j, i):
            keep[j] = False
```

Note:
Rows in diagram correspond to i, columns correspond to j
Each box represents a comparison between box i and box j

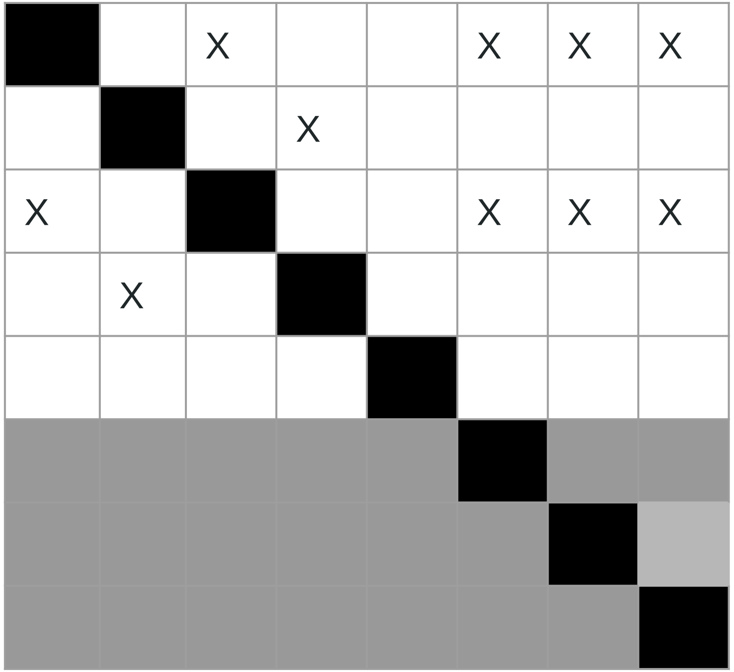


Serial Code: Unordered version

We can still discard some comparisons, but this algorithm isn't guaranteed to be as efficient as the ordered algorithm

Initially, we believe we should keep every box. The boxes are ordered by their detection probabilities. (Boxes[0] has highest probability)

```
for i if keep[i] == True:
    for j if keep[j] == True:
        R = iou_compare(j, i)
             if R > threshold:
         if prob(j) > prob(i):
             keep[j] = False
         else:
             keep[i] = False
```



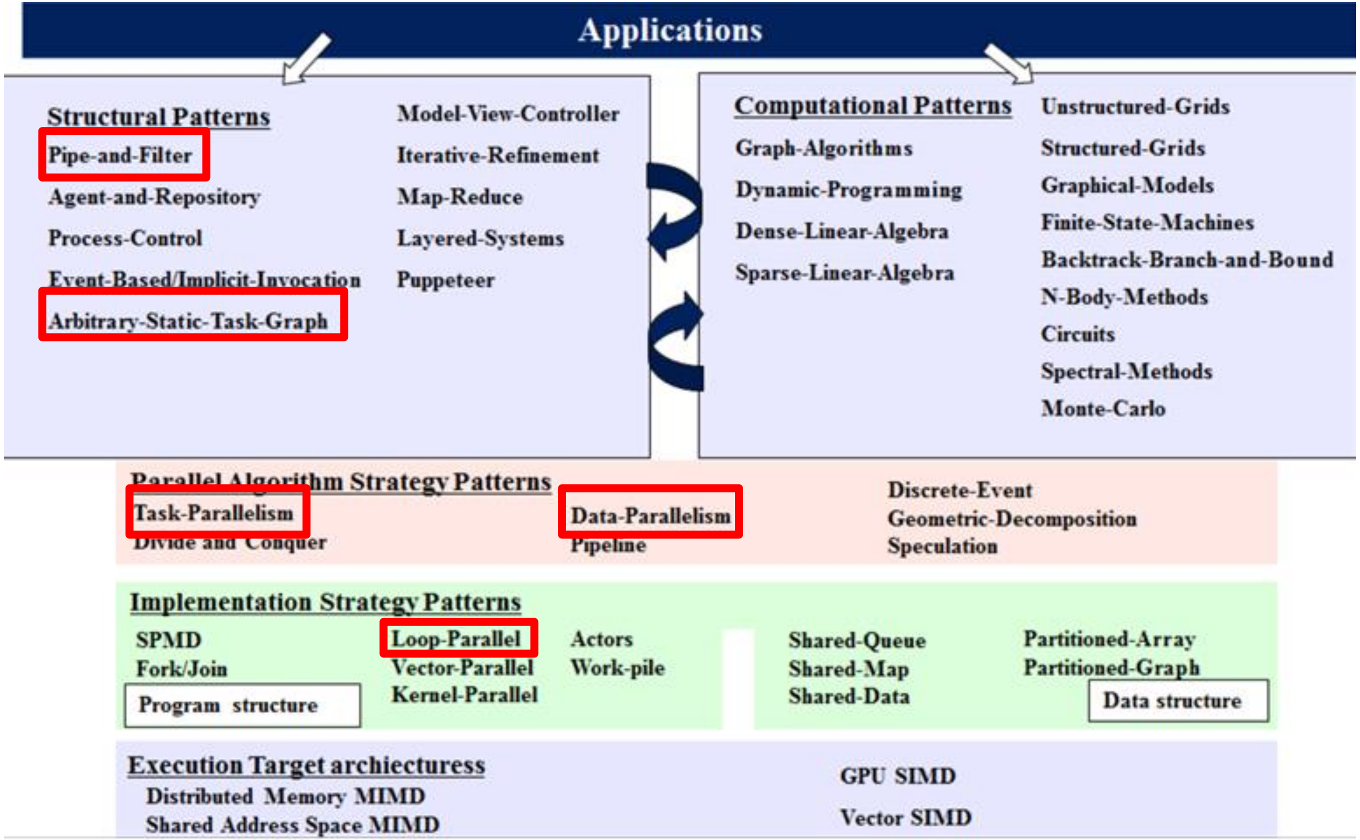
This version is not 100% accurate to the serial version, but runs faster

No longer need to sort the boxes by probabilities

Overview

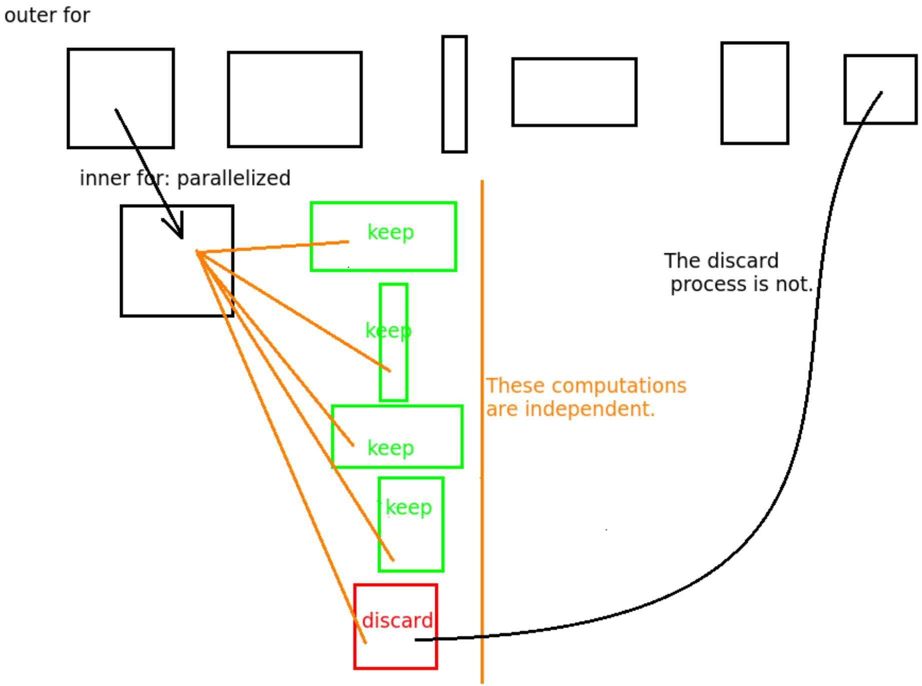
- Project Description
- Algorithm
- Serial Code
- **Parallelization**
- Results

Parallelization: OMP



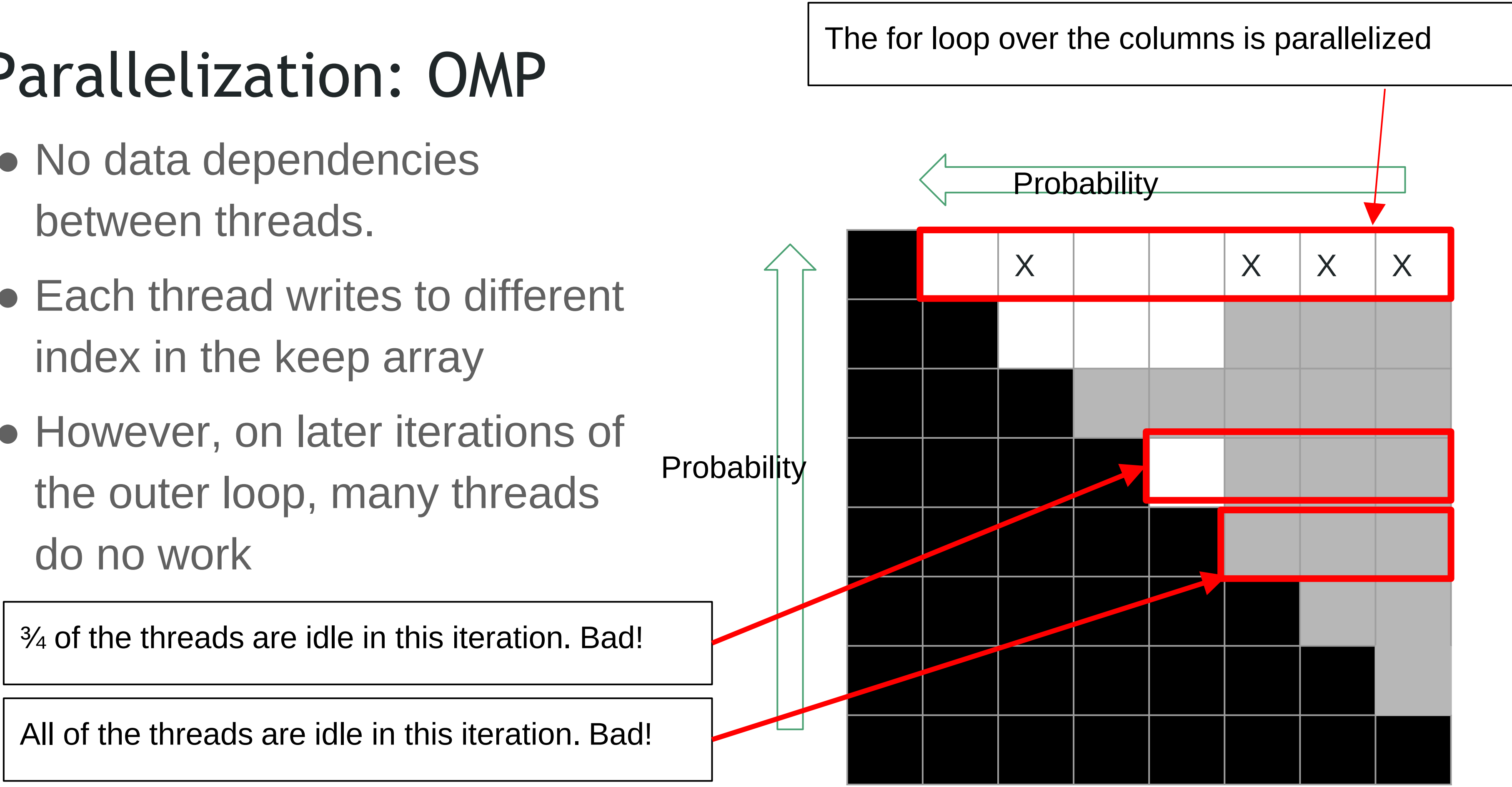
Parallelization: OMP

- Inner loop only, because we're throwing out boxes as we go along
- All threads compare box i to a unique box j
  - Threads mark j as discarded
- Data parallel: j boxes in organized array, each thread process unique j
- Task parallel: Each comparison is an independent task
- 2x speedup over serial c
  - Many threads are idle in later iterations



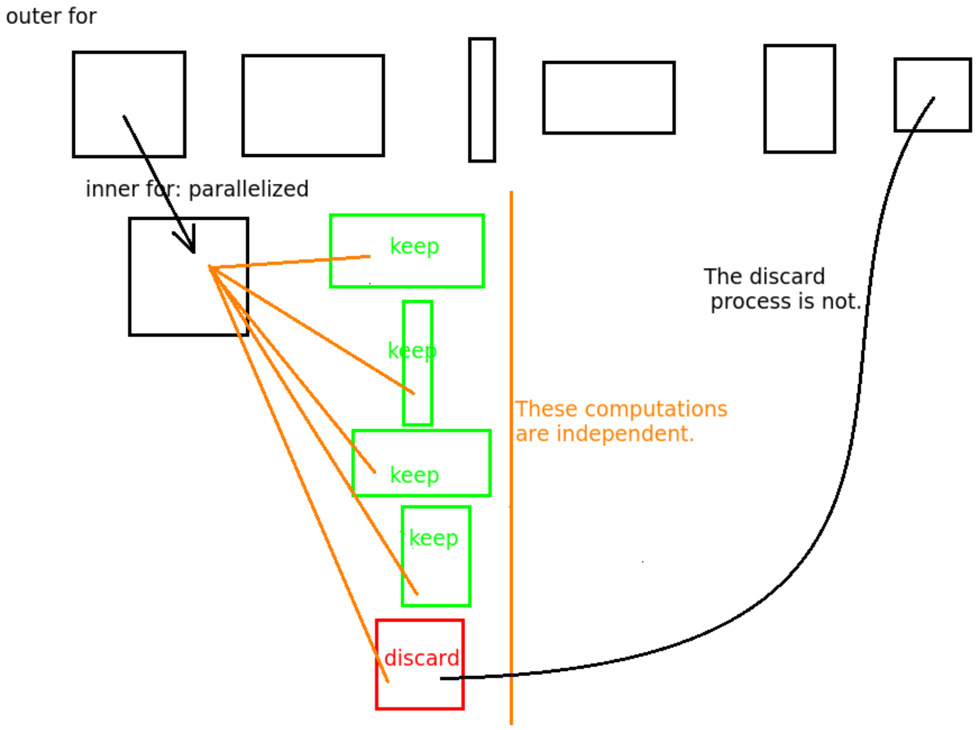
Parallelization: OMP

- No data dependencies between threads.
- Each thread writes to different index in the keep array
- However, on later iterations of the outer loop, many threads do no work



Parallelization: OMP unordered

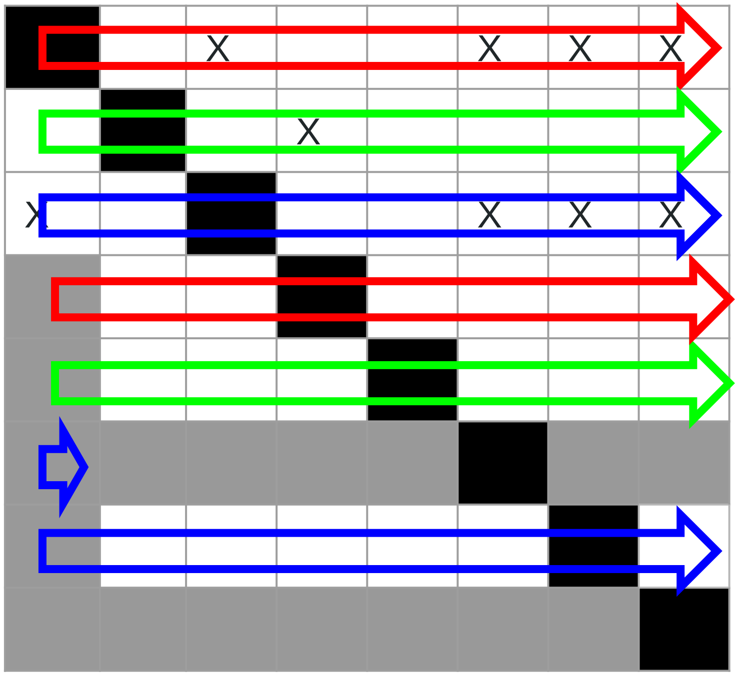
- Parallelize the outer for loop of th
unordered algorithm
- ~7x speedup compared to serial
- Not entirely accurate, since we a
not going strictly by order of
probability
- Still task and data parallel
    - Data parallel: we iterate over an **unsorted** array of boxes now
    - Task parallel: each comparison is unique



Parallelization: OMP unordered

- Parallelize the outer for loop
- Threads do not need to communicate, since they perform all possible computations
- Later unnecessary rows are skipped completely
  - Compare to previous implementation

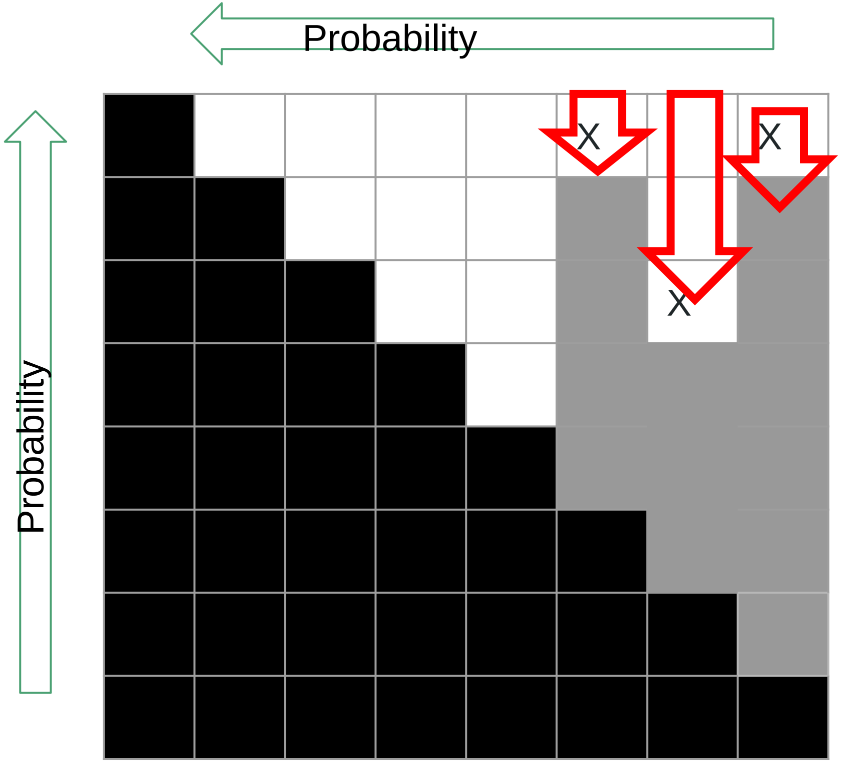
Ex: Three rows in parallel



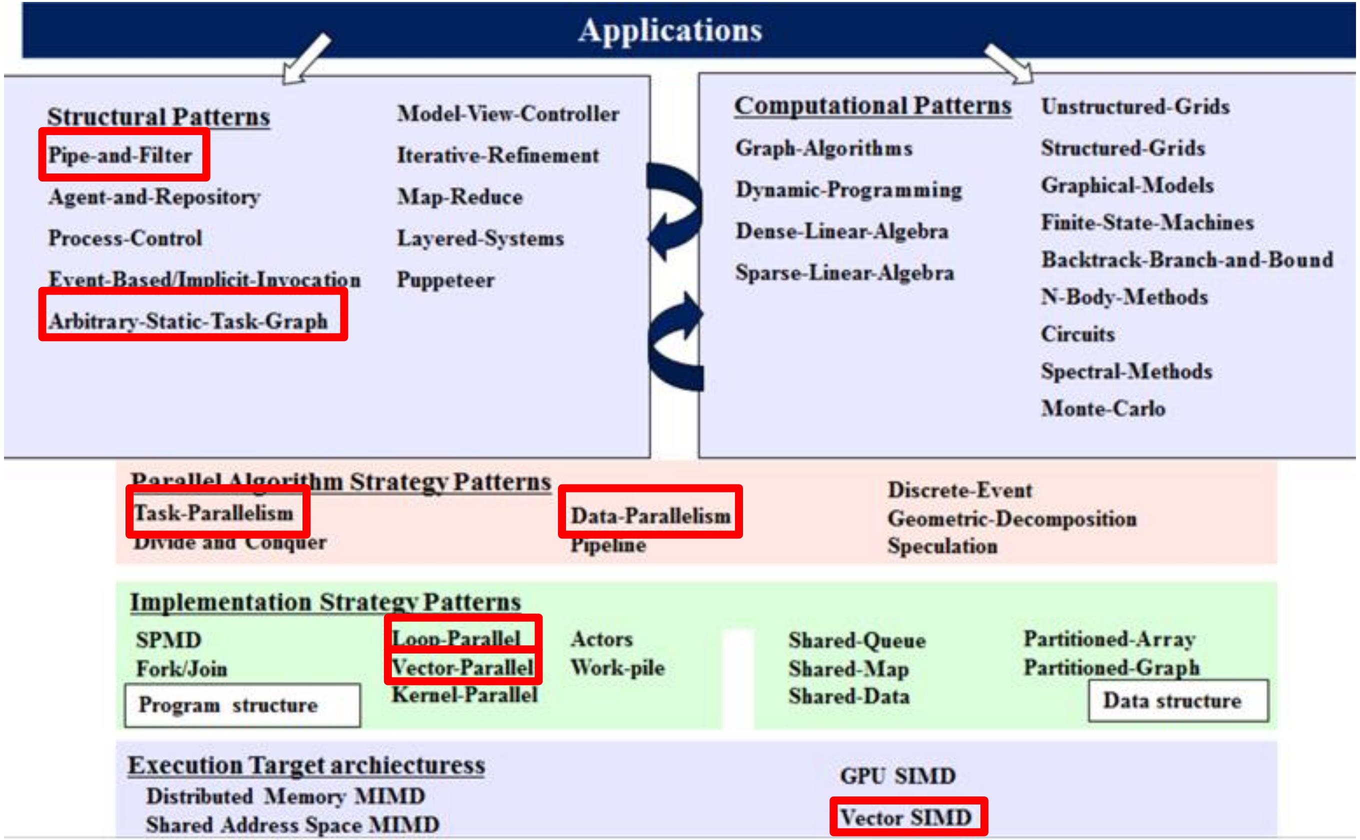
Dynamic scheduling lets threads move on to next row. Ex: Blue thread sees to skip row 6, is rescheduled to row 7

Parallelization: OMP Alternate

- Parallelize the columns of the serial algorithm, starting from low probability boxes
- No interthread communication, threads all write to unique position in keep array
- Slow, many excess computations are performed. 0.2x as fast as serial c
    - We must compute down every column
- Task parallel: Seeing if we want to keep box X is a task
- Data parallel: Boxes still in a sorted array

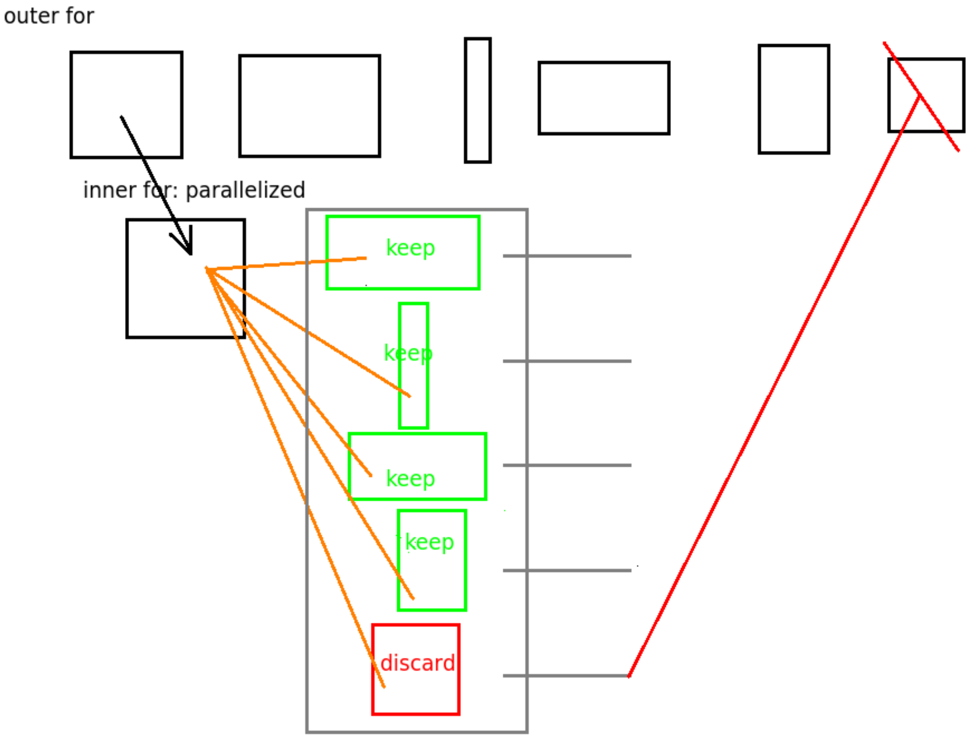


Parallelization: SIMD



Parallelization: SIMD

- Vectorized the individual iou comparisons among boxes
- Gather and compute ratios in batches, then compare ratios and discard in groups
- Used alongside OMP
    - 9x speedup compared to serial C
    - No error unlike unordered C



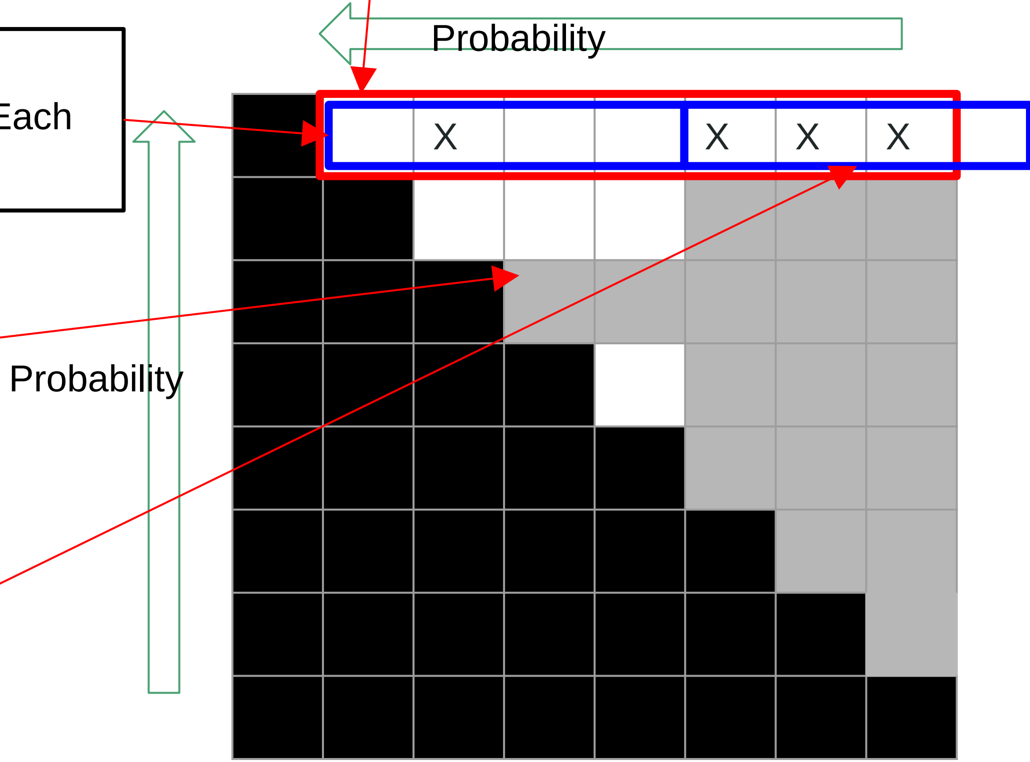
Parallelization: SIMD

Computation down rows (outer for loop) is serial

Comparisons and keep decisions are performed in batches of 8 using AVX. Each set of 8 is handed to own thread

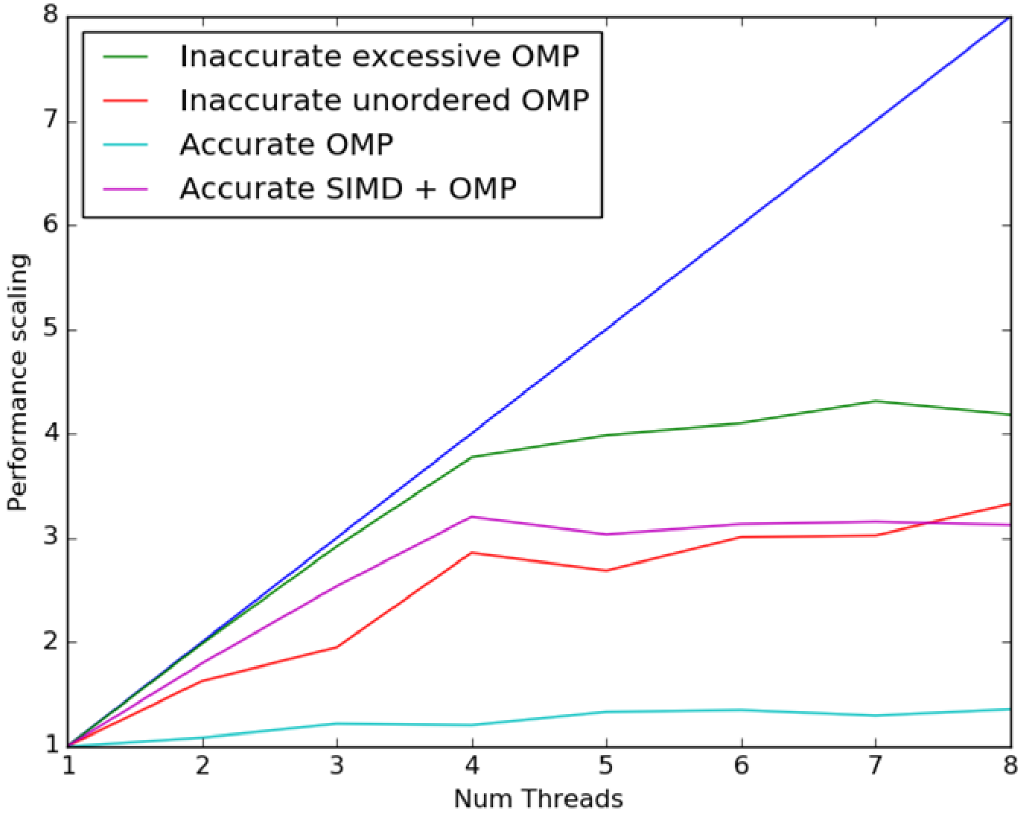
Similarity to serial version lets us skip many rows. Both computationally efficient and parallelized!

Padded array lets us reference off the edge safely. Lets us keep the work in one for loop

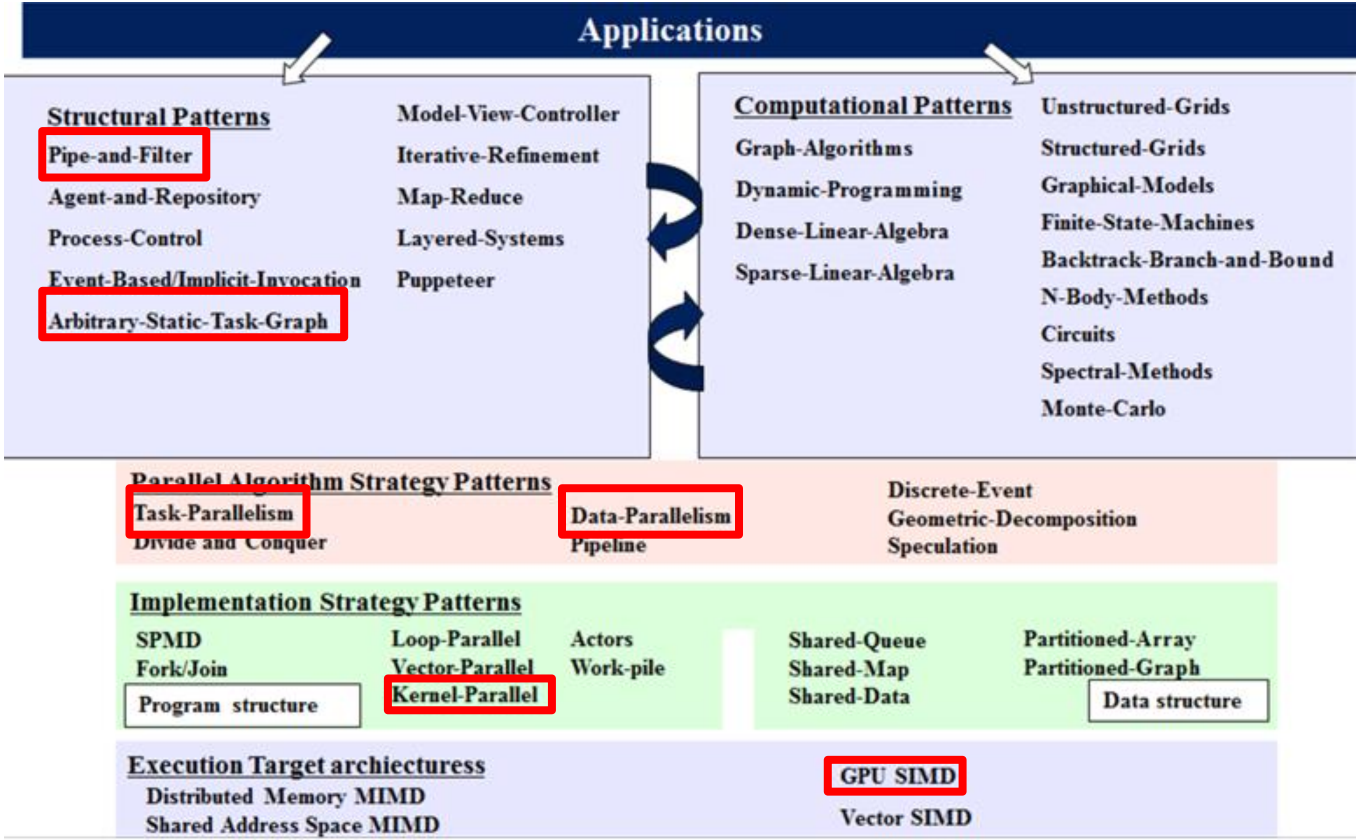


Scaling Plot

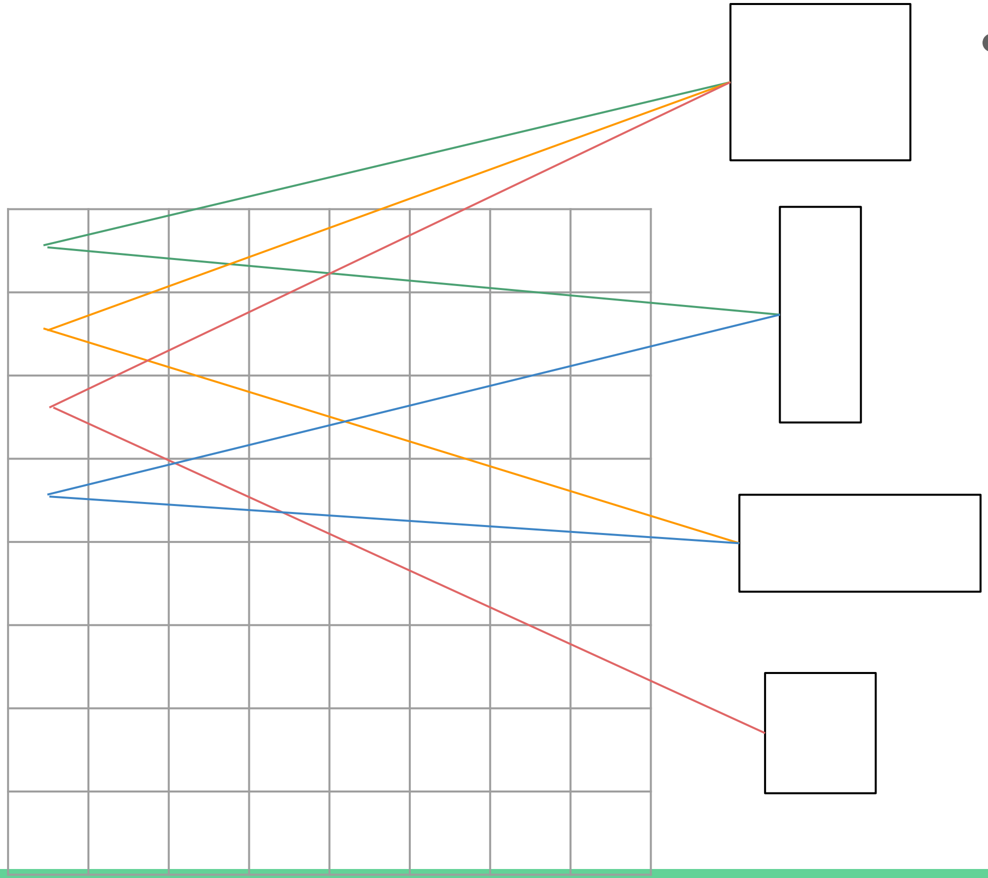
As expected, drop off of scaling after 4 threads due to availability of only 4 physical cores.



Algorithm: Patterns



Parallelization: GPU



- Each work item calculates the IoU of only one pair of bounding boxes
- Discard process is omitted, it requires inter-workitem communication
- Result is slow because of overhead - has to:
  - set up the kernel
  - load data to the GPU
  - copy result back to the host
  - 3x speedup (ignoring setup time)
- Results agree with the ICASSP paper. (See references)

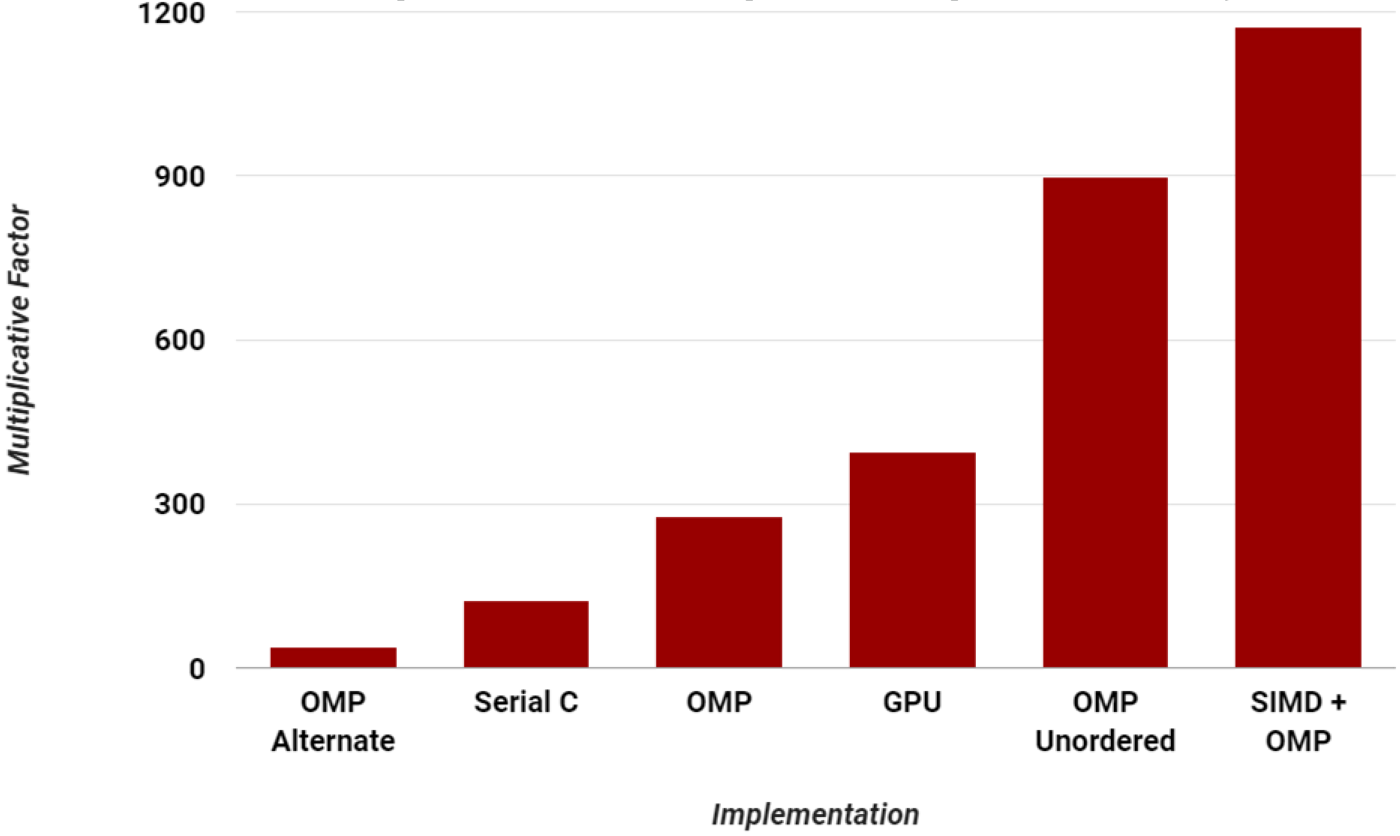
Overview

- Project Description
- Algorithm
- Naive Code
- Parallelization
- **Results**

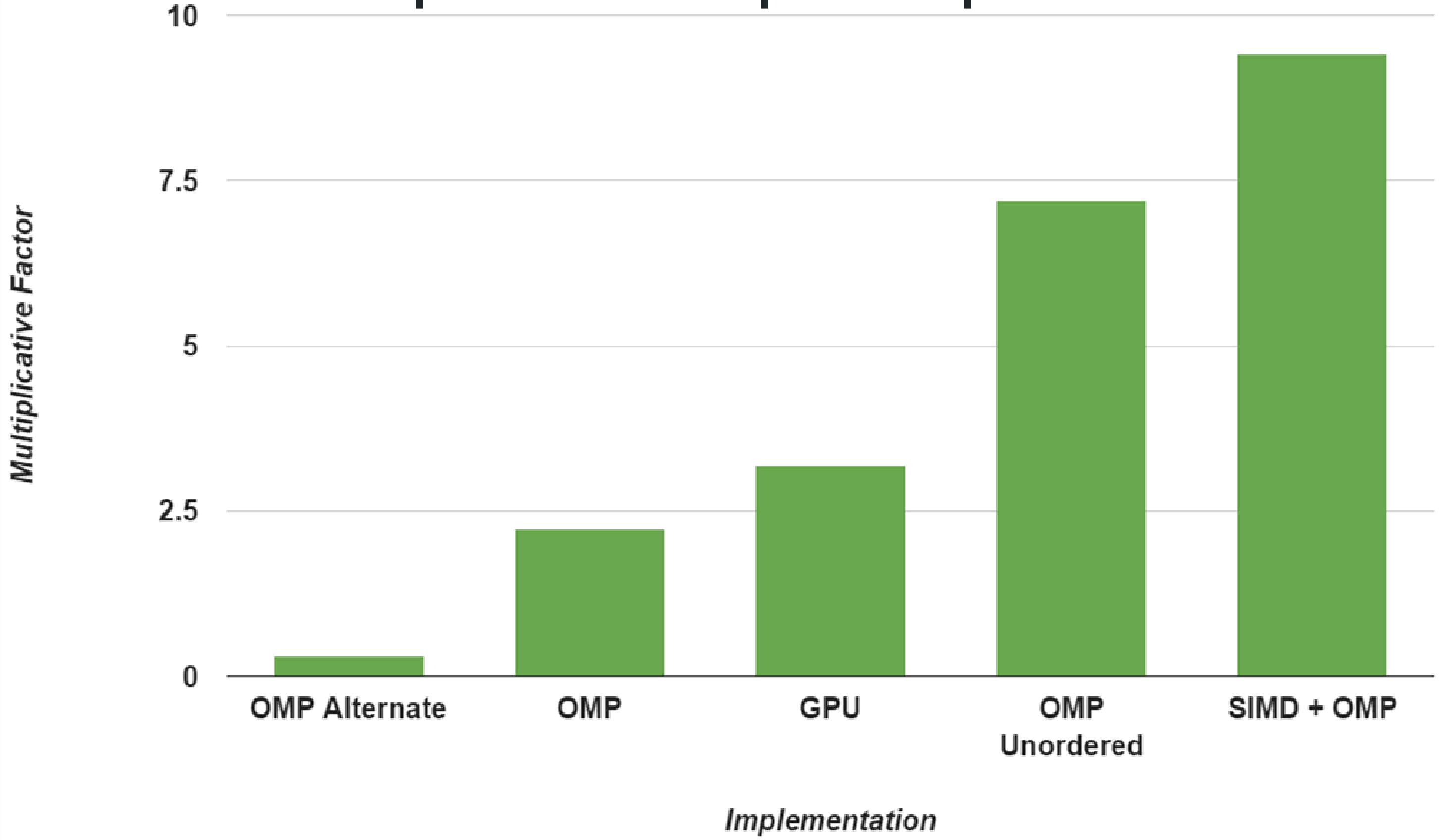
Results: Speedup (100 images, 15000 boxes/image)

Implementation	Total Time (s)	Speedup (x serial C)	Speedup (x serial Python)
Serial Python	553.949	0.008	1.000
Serial C	4.446	1.000	124.6
OMP	1.987	2.237	278.7
OMP Unordered	0.618	7.196	896.6
OMP Alternate	14.852	0.299	37.30
SIMD + OMP	0.472	9.414	1173
GPU	1.397	3.182	396.5

Results: Multiplicative Speedup over Python

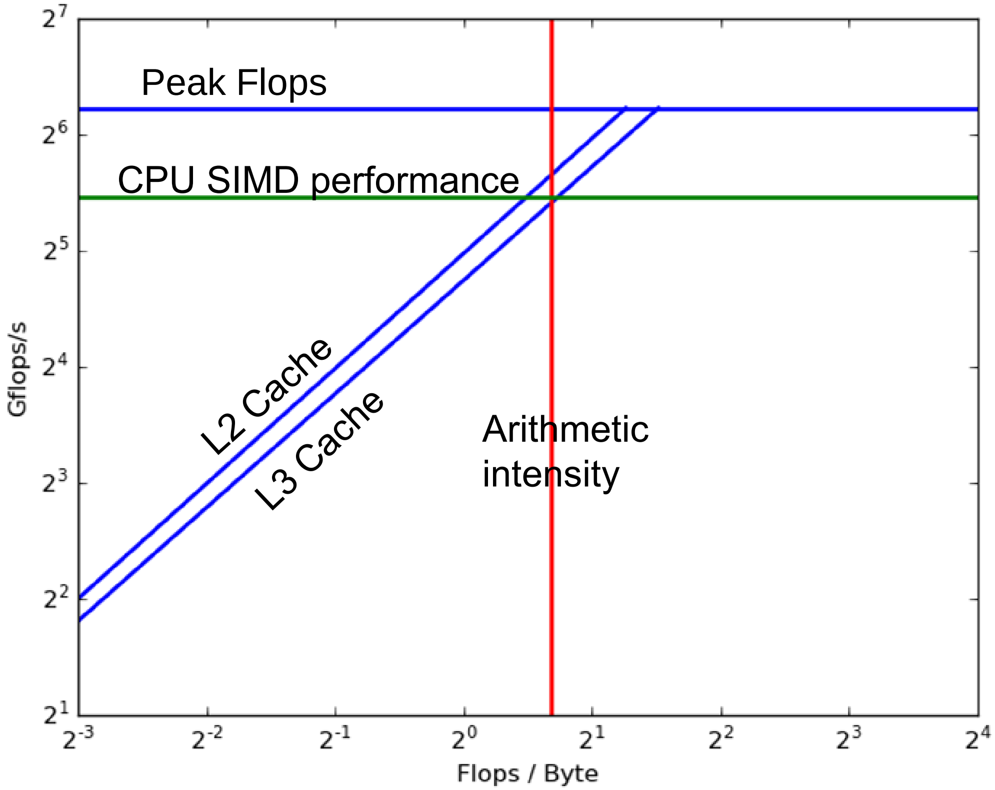


Results: Multiplicative Speedup over C



Roofline Analysis

- 18% away from peak performance, given our algorithm
- Arithmetic Intensity
= 32/20 = 1.6 Flops/Byte
- CPU SIMD Performance
= 43.78 GFlops/s
- Possible L3 Performance
= 41.6 GFlops/s
- Possible L2 Performance
= 52.8 GFlops/s



Results: Speedup

- We started with something that took >5min for 100 images, or >3s per image
- Not feasible, when a car camera is going at 60fps = 1 frame/12 ms
- We cut the time down to ~5ms per image!
- This makes using NMS on real-time image processing feasible, possibly on multiple cameras! Cool!

More Thoughts

- Our unordered algorithm performed slightly better than the traditional algorithm, at the cost of accuracy
  - 0.19% error compared to serial code
  - Error comes from bounding boxes w/ equal probabilities
  - More work is needed to see if this error is acceptable
- SIMD+OMP was logically equivalent to Bichen’s code; 0% error
  - Error calculated by comparing bounding box output list for a test dataset between our function and Bichen’s serial python function
- GPU NMS may still be feasible
  - Since many computer vision methods run on GPUs, the CPU to GPU memory transfer may be avoided

Resources/References

- Neubeck, Alexander, and Luc Van Gool. "Efficient non-maximum suppression." *18th International Conference on Pattern Recognition (ICPR'06)*. Vol. 3. IEEE, 2006. https://pdfs.semanticscholar.org/52ca/4ed04d1d9dba3e6ae30717898276735e0b79.pdf
- Oro, David, et al. "Work-efficient parallel non-maximum suppression for embedded GPU architectures." *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2016. http://ieeexplore.ieee.org/iel7/7465907/7471614/07471831.pdf
- http://www.pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/
- http://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/