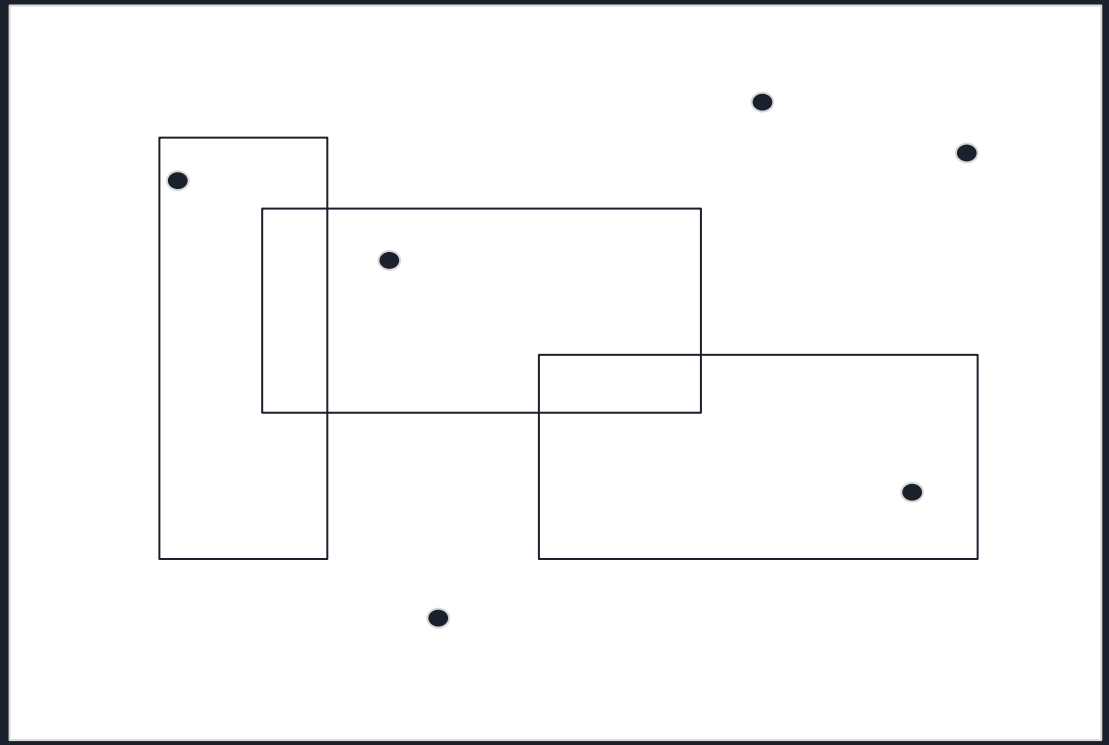
DPP Examination

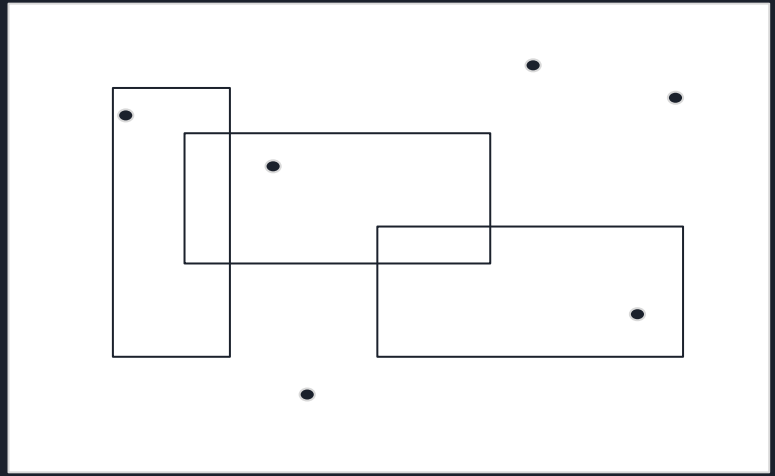
Alexandre DA CUNHA--GUIBORAT (vdp873)
Oleksandr KRYSHTALOV (MDQ842)

RangeQuery2d



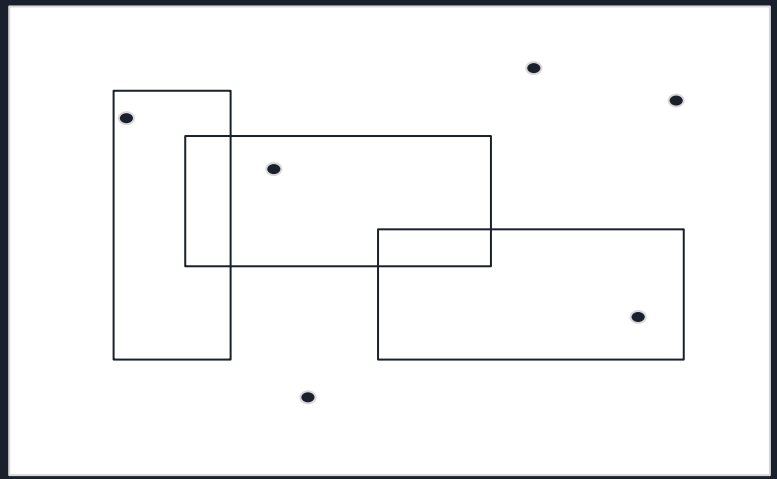
Naive Sequential Implementation

```
type Point = {x : f64, y : f64}
type Rectangle = {ll : Point, ur : Point}
type Cell [n] = {rectangle : Rectangle, p_in : [n]i64}

def within (p:Point) (r:Rectangle) : bool =
    p.x <= r.ur.x &&
           p.x >= r.ll.x &&
           p.y <= r.ll.y &&
           p.y >= r.ur.y


def rangeQuery2d [n] [m] (rects: [m]Rectangle)  (points: [n]Point): [m]i64 =
    let counts = replicate m 0
    let res = loop (new_counts, r) = (counts, 0) for r < m do
        let (count, _) = loop (acc, p) = (0, 0) for p < n do
        -- loops flipped around for better temporal locality
            if within points[p] rects[r]
                then (acc+1, p+1)
                else (acc, p+1)
        in
        (new_counts with [r] = count, r+1)
    let (final_counts, _) = res
    in final_counts
```



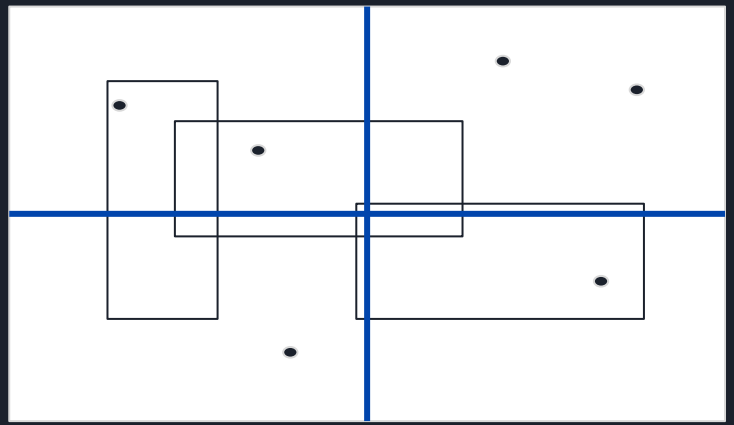
Naive Parallel Implementation

```
type Point = {x : f64, y : f64}
type Rectangle = {ll : Point, ur : Point}
type Cell [n] = {rectangle : Rectangle, p_in : [n]i64}

def within (p:Point) (r:Rectangle) : bool =
    p.x <= r.ur.x &&
           p.x >= r.ll.x &&
           p.y <= r.ll.y &&
           p.y >= r.ur.y

-- counts the number of points in the rect
def points_in_rect [n] (points: [n]Point) (rect: Rectangle): i64 =
    reduce (+) 0 (map (\p -> if within p rect then 1 else 0) points)

def rangeQuery2d [n] [m] (rects: [m]Rectangle)  (points: [n]Point): [m]i64 =
    map (points_in_rect points) rects
```



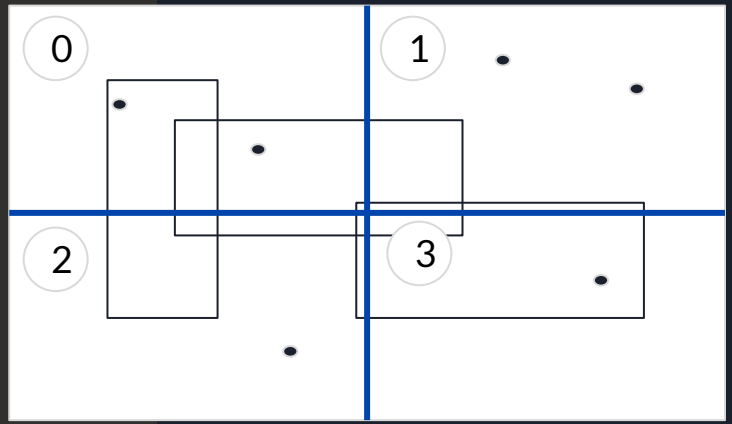
Grid Parallel Implementation

```
def rangeQuery2d_grid_bis [m] [n] (depth : i64) (rectangles : [m]Rectangle) (points : [n]Point) : [m]i64 =

    let d = depth
    let cells = preprocess_create_grid_depth d points


    let solution =  map(\r ->
                        let cells_to_consider : []Cell[n] = get_subarray_cells_bis r cells
                        let subarray_pts : []Point = get_subarray_point_bis cells_to_consider points
                        in nb_points_in_rectangle r subarray_pts
                    ) rectangles
    in solution
```



Preprocessing

```
-- Given a depth and index of a cell <c>, returns c.rectangle
def cell_helper_rectangle_depth (depth : i64) (ind : i64) : Rectangle =

    let d = f64.i64 depth
    -- let i = f64.i64 ind
    let u = 1/(2**d)

    let x_subdiv = ind % (2**depth)
    let y_subdiv = ind / (2**depth)

    let fx_subdiv = f64.i64 x_subdiv
    let fy_subdiv = f64.i64 y_subdiv

    let ll : Point = {x = fx_subdiv * u, y = (fy_subdiv + 1) * u}
    let ur : Point = {x = (fx_subdiv + 1)* u, y = fy_subdiv * u}
    let r : Rectangle = {ll = ll, ur = ur}
    in r


-- This function returns a grid of cells for a given depths
def preprocess_create_grid_depth [n] (depth : i64) (points : [n]Point) : []Cell [n] =

    let rect_grid : []Rectangle = map(\i -> cell_helper_rectangle_depth depth i) (iota (4**depth) )
    let p_in_grid : [][n]i64 = map(\r -> points_in_rectangle r points) rect_grid
    let grid : []Cell [n] = map2(\r p -> {rectangle = r, p_in = p}) rect_grid p_in_grid
    in grid
```



Optimisation

```
-- This function returns an array with the number of points in the rectangle at this index
def rangeQuery2d_grid [m] [n] (depth : i64) (rectangles : [m]Rectangle) (points : [n]Point) : [m]i64 =

    let d = depth
    let cells = preprocess_create_grid_depth d points


    let solution =  map(\r ->
                        let subarray_pts : []Point = get_subarray_point r cells points -- This function must be optimized to make it quick
                        in nb_points_in_rectangle r subarray_pts
                    ) rectangles
    in solution
```

```
-- Lifted operator to make the operation of gattering all point in c1 union c2
-- This way, we can compute that in //
-- Returns a cell with a dummy rectangle and a p_in = c1.p_in \cup c2.p_in
def points_in_cells_reduce [n] (c1 : Cell [n]) (c2 : Cell [n]) : Cell [n] =

    let r_ll = { x = 0, y = 0}
    let r_ur = { x = 0, y = 0}
    let r = {ll = r_ll, ur = r_ur}

    let p_in = map2(\c1_p c2_p -> if c1_p == 1 || c2_p == 1 then 1 else -1) c1.p_in c2.p_in

    let c = {rectangle = r, p_in = p_in}
    in c

-- This function returns an array of flag with a 1 at each points in \cup cs, -1 otherwise
def points_in_cells [n][m] (cs : [m]Cell [n]) : [n]i64 =

    let r_ll = { x = 0, y = 0}
    let r_ur = { x = 0, y = 0}
    let r = {ll = r_ll, ur = r_ur}
    let ps_init = map(\_ -> -1) (iota n)
    let c_neutral = {rectangle = r, p_in = ps_init}

    let custom_cell = reduce (points_in_cells_reduce) c_neutral cs
    let ps_in = custom_cell.p_in
    in ps_in
```

```
-- This function returns an array with 1 when a cell is crossed by r, -1 otherwise
def rect_cross_cells [n][m] (r : Rectangle) (cs : [m]Cell [n]) : [m]i64 =
    let selected_cells = map(\c -> rect_cross_cell r c ) cs
    in selected_cells


-- This function turns an flag_array of cells into an array of cells
def cells_flags_2_cells_array [n][m] (cs_flag : [m]i64) (grid : [m]Cell [n]) : []Cell [n] =
    let (_, subgrid) = unzip (filter(\(f,_) -> f>0) (zip cs_flag grid))
    in subgrid

-- This function returns an array of Point, where points flag[i] == 1
def points_flags_2_points_array [n] (points_flags : [n]i64) (P : [n]Point) : []Point =

    let (_, subpoints) = unzip (filter(\(f,_) -> f>0) (zip points_flags P))
    in subpoints


-- This function returns the subarray of points to consider to then brute force
def get_subarray_point [n][m] (r : Rectangle) (grid : [m]Cell [n]) (P : [n]Point) : []Point =

    let cells_flags_to_consider = rect_cross_cells r grid
    let cells_to_consider = cells_flags_2_cells_array cells_flags_to_consider grid

    let points_flags_to_consider = points_in_cells cells_to_consider
    let subarray_point : []Point = points_flags_2_points_array points_flags_to_consider P

    in subarray_point
```

Optimisation (2)

```
def _get_subarray_cells_bis [n][m] (r : Rectangle) (grid : [m]Cell [n]) : []Cell [n] =


    let cells_flags_to_consider = rect_cross_cells r grid

    let cells_scatter_sz_helper = map(\f -> if f == 1 then 1 else  0) cells_flags_to_consider
    let cells_scatter_sz = reduce (+) 0  cells_scatter_sz_helper

    let scatter_idx_offset = scan (+) 0 cells_scatter_sz_helper
    let scatter_idx = map(\i -> i - 1 ) scatter_idx_offset
    let scatter_idx_masked = map2(\f i -> if f == 1 then i else -1 ) cells_flags_to_consider scatter_idx

    let dest = map(\_ -> grid[0]) (iota cells_scatter_sz)
    let cells_to_consider = scatter dest scatter_idx_masked grid

    in cells_to_consider


-- This function returns the subarray of points to consider to then brute force
def _get_subarray_point_bis [n][m] (cells : [m]Cell [n]) (P : [n]Point) : []Point =

    let points_flags_to_consider = points_in_cells cells

    let points_scatter_sz_helper = map(\f -> if f == 1 then 1 else  0) points_flags_to_consider
    let points_scatter_sz = reduce (+) 0  points_scatter_sz_helper

    let scatter_idx_offset = scan (+) 0 points_scatter_sz_helper
    let scatter_idx = map(\i -> i - 1 ) scatter_idx_offset
    let scatter_idx_masked = map2(\f i -> if f == 1 then i else -1 ) points_flags_to_consider scatter_idx

    let dest = map(\_ -> P[0]) (iota points_scatter_sz)
    let subarray_point = scatter dest scatter_idx_masked P

    in subarray_point
```

Performance

Memory allocation issue.

On smaller data :

```
benchmark_all.fut:bench_simple_parallel (no tuning file):
2DinCube (5k):        322µs (95% CI: [     310.9,      338.8])

benchmark_all.fut:bench_point_grid (no tuning file):
2DinCube (5k):     241725µs (95% CI: [  239974.0,   242729.7])

benchmark_all.fut:bench_point_grid_bis (no tuning file):
2DinCube (5k):      35760µs (95% CI: [   35124.1,    36173.6])
```

```
PBBS times (s)
2DinCube (1M):     '0.583', '0.584', '0.58', geomean = 0.583
2Dkuzmin (1M):     '0.582', '0.584', '0.578', geomean = 0.581
2DinCube (10M):    '9.737', '9.813', '9.845', geomean = 9.798
2Dkuzmin (10M):    '10.15', '10.673', '10.578', geomean = 10.465
```

Improvement

Cells may store array of different length -> Overhead

Change for a quadtree implementation -> Sequential / Massive usage of memory

Conclusion