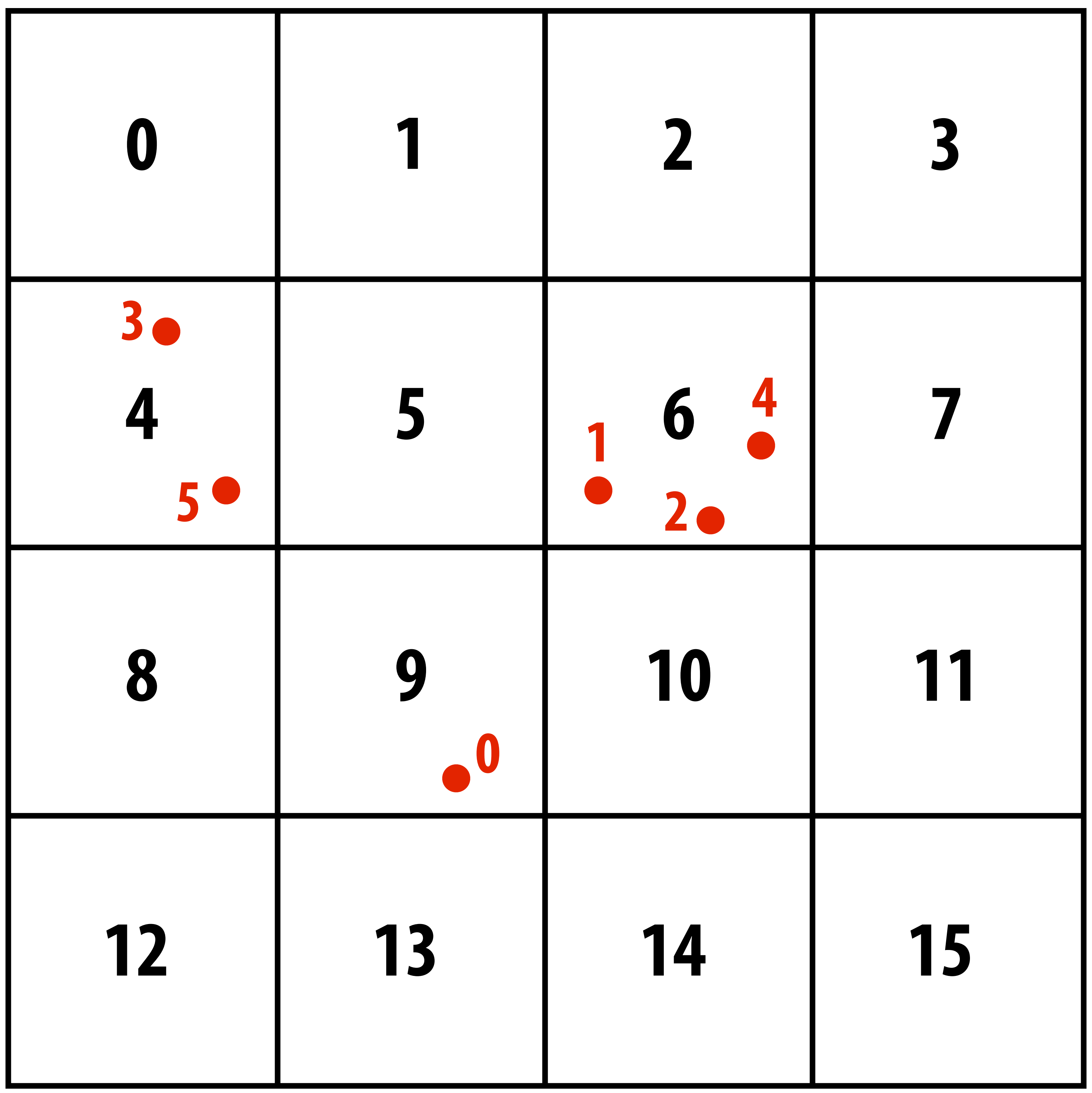
Lecture 8:
Parallel Programming Case Studies

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Contention example (last time)

- Problem: place many (e.g., 100K) point particles in a 16 cell uniform grid
  - Parallel data structure manipulation problem: build a grid of lists
- Recall: 15 cores, up to 1024 threads per core on GTX 480 GPU



Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

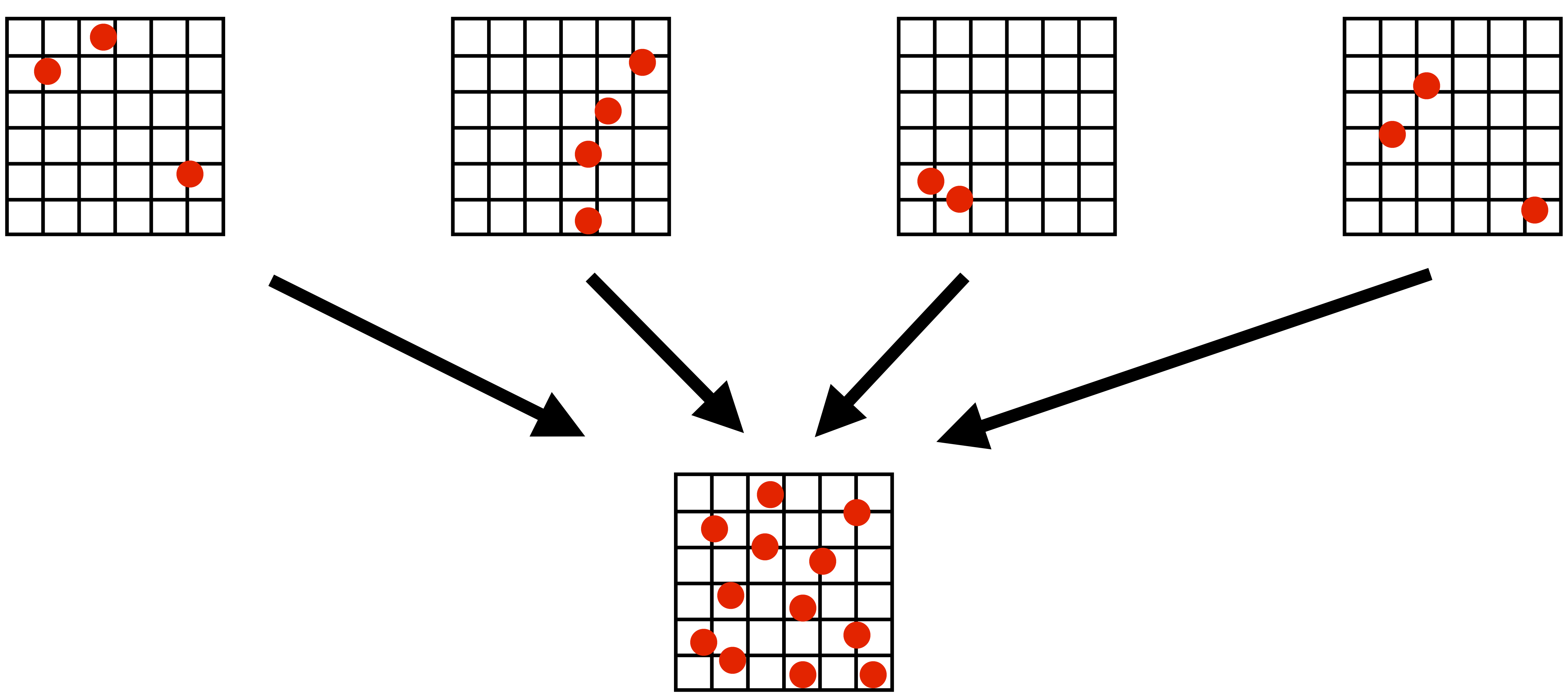
Contention example

- First answer from last time: partition work by cells. For each cell, independently compute overlapping particles
  - 16 parallel tasks (insufficient parallelism: need thousands of independent tasks)
  - Also: performs 16 times more particle-in-cell computations than sequential algorithm (it's no faster)

- Another answer: assign one particle to each CUDA thread. Compute cell containing particle. Atomically update list.
  - Massive contention: thousands of threads contending to update 16 lists
  - Also: how are you going to "update the list"?

Contention example

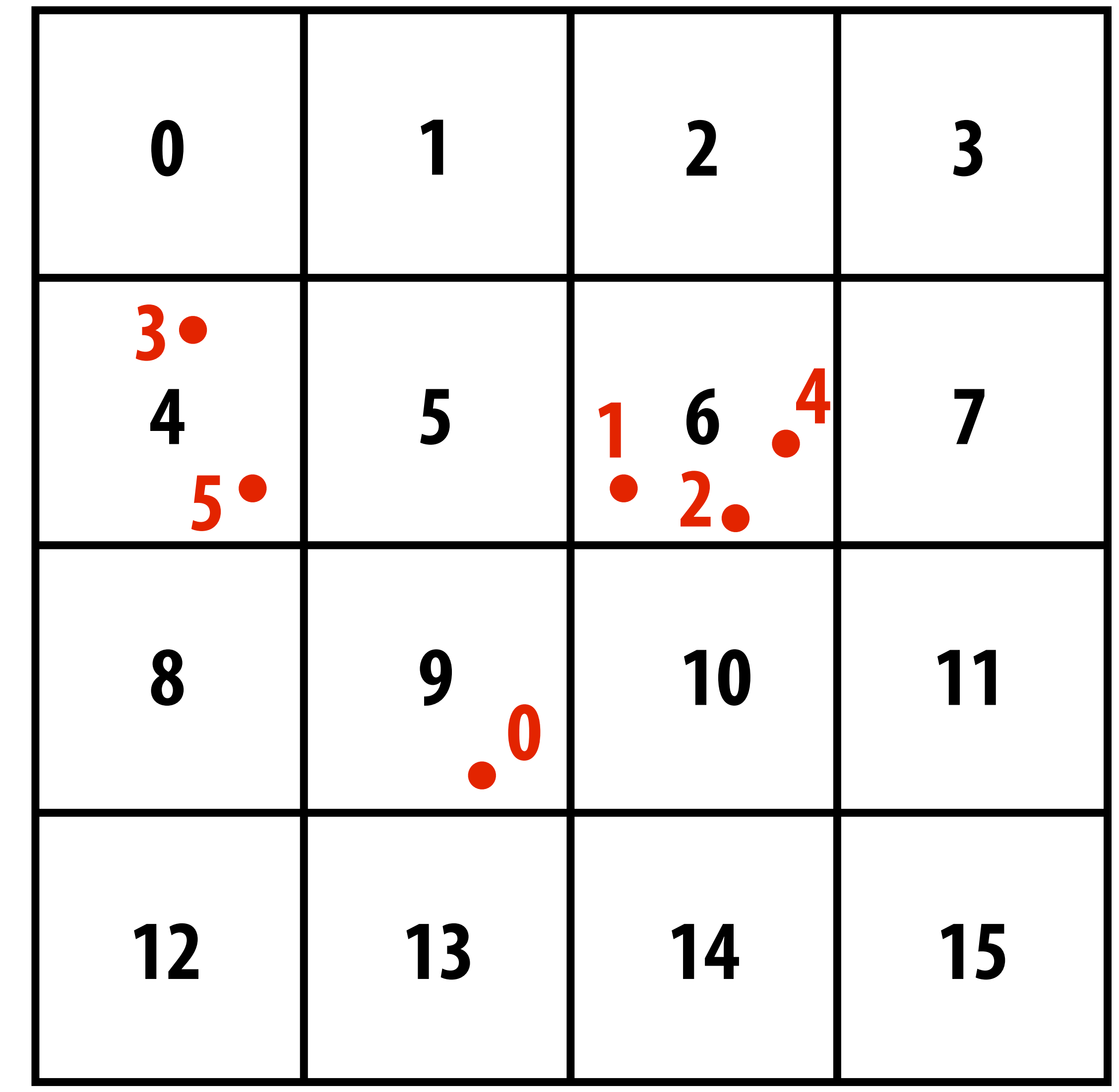
- Yet another answer: generate N grids in parallel, each thread updates one of the grids.
  - Example: create 15 grids on GTX 480 (one per core)
  - All threads assigned to core update same grid
    - Faster synchronization: contention reduced by factor of N, performed on local variables
  - Extra work: merging the grids at the end of the computation



Data-parallel solution

Step 1: compute cell containing each particle

0	1	2	3	4	5
9	6	6	4	6	4



Step 2: sort by cell

3	5	1	2	4	0
4	4	6	6	6	9

Step 3: find start/end of each cell

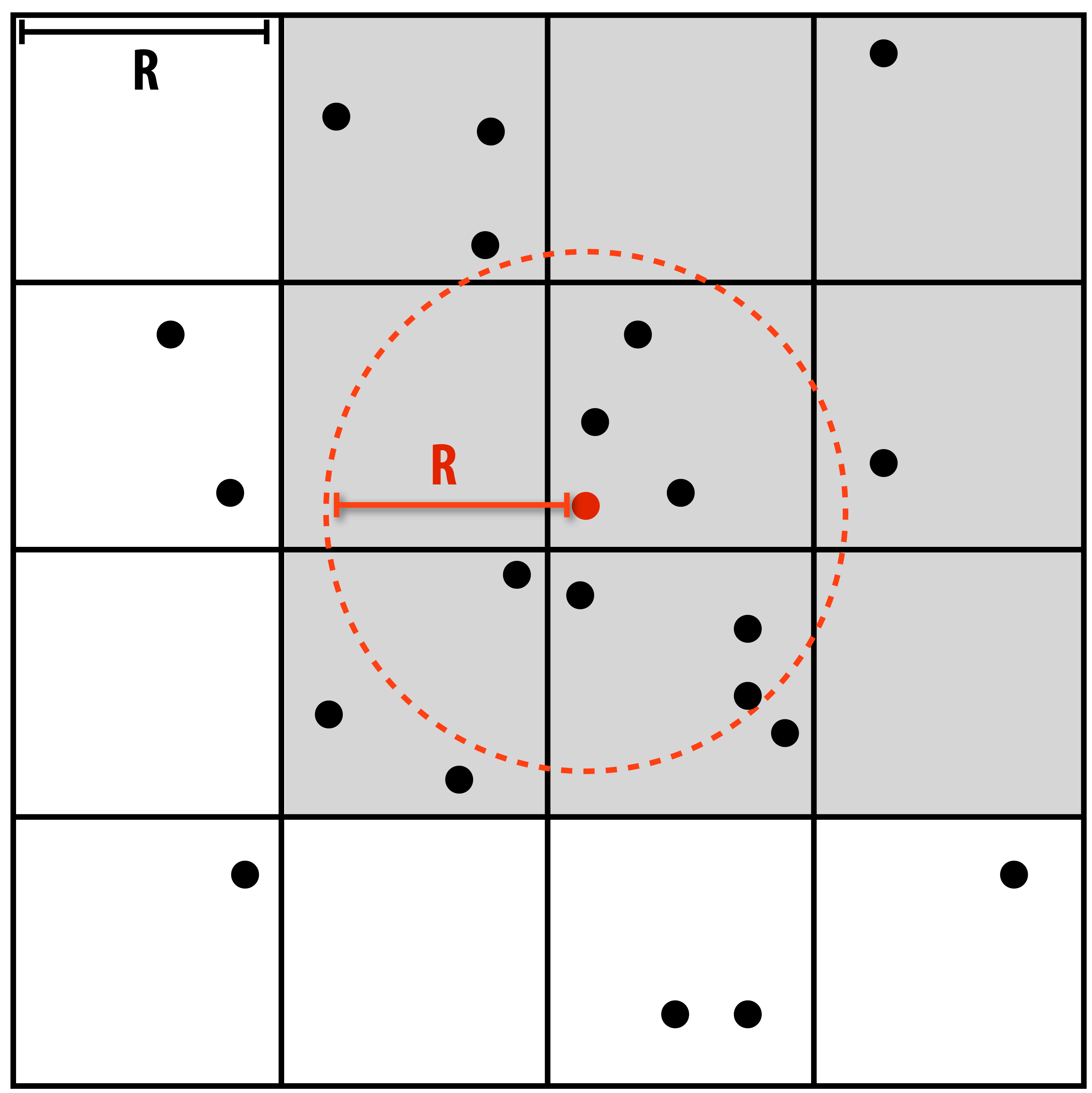
```
cell = A[index]
if (index == 0 || a[index] != a[index-1]) {
    cell_starts[cell] = index;
    cell_ends[A[index-1]] = index;
}
if (index == numParticles-1) // special case for last cell
    cell_ends[cell] = index+1;
```

Removes need for fine-grained synchronization at cost of sort and extra passes over the data (extra BW)

cell_starts	0xff	0xff	0xff	0xff	0	0xff	2	0xff	0xff	5	0xff	. . .
cell_ends	0xff	0xff	0xff	0xff	2	0xff	5	0xff	0xff	6	0xff	. . .
	0	1	2	3	4	5	6	7	8	9	10	

Common use: N-body problems

- Common operation is to compute interactions with neighboring particles
- Example: find all particles within radius R
  - Create grid with cells of size R
  - Only need to inspect particles in surrounding grid cells



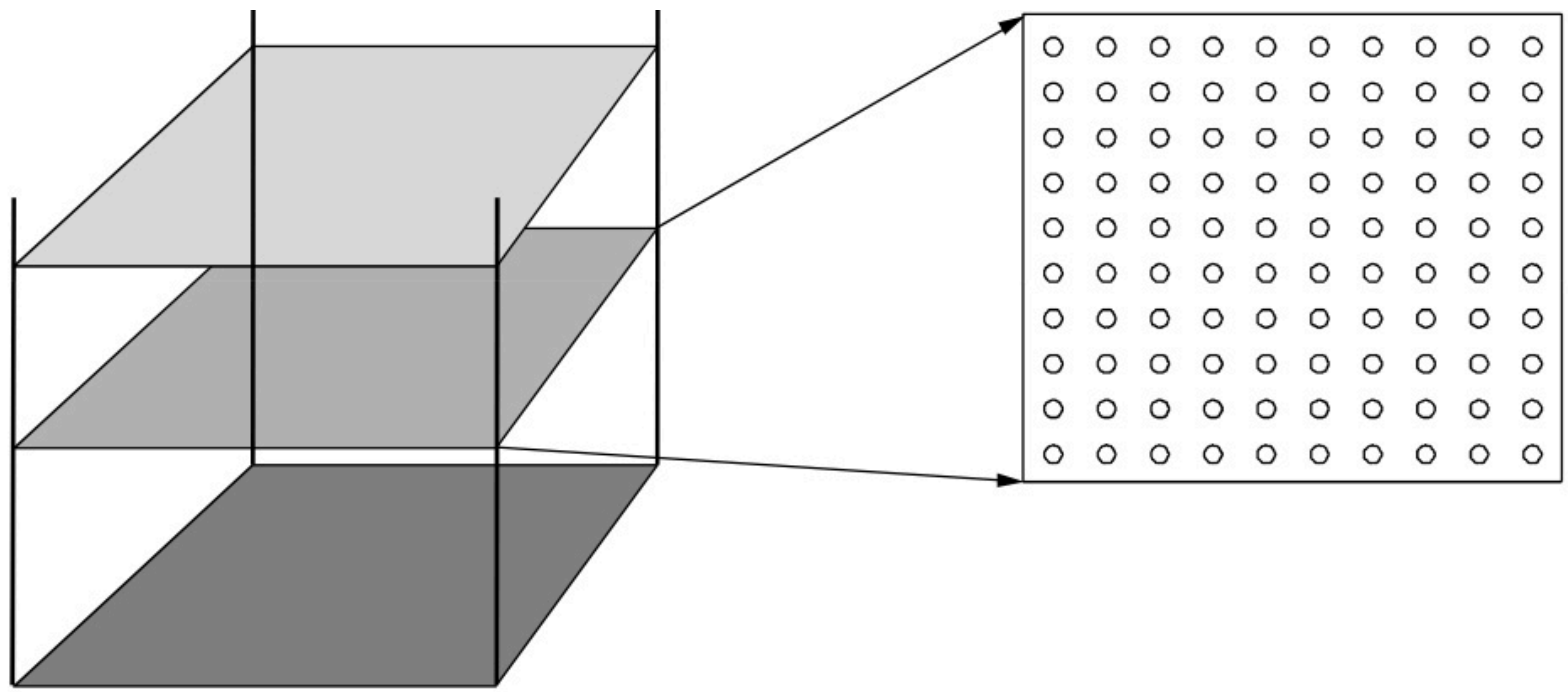
Today

- Parallel application case studies!

- Three examples
  - Ocean
  - Galaxy simulation (Barnes-hut)
  - Ray tracing

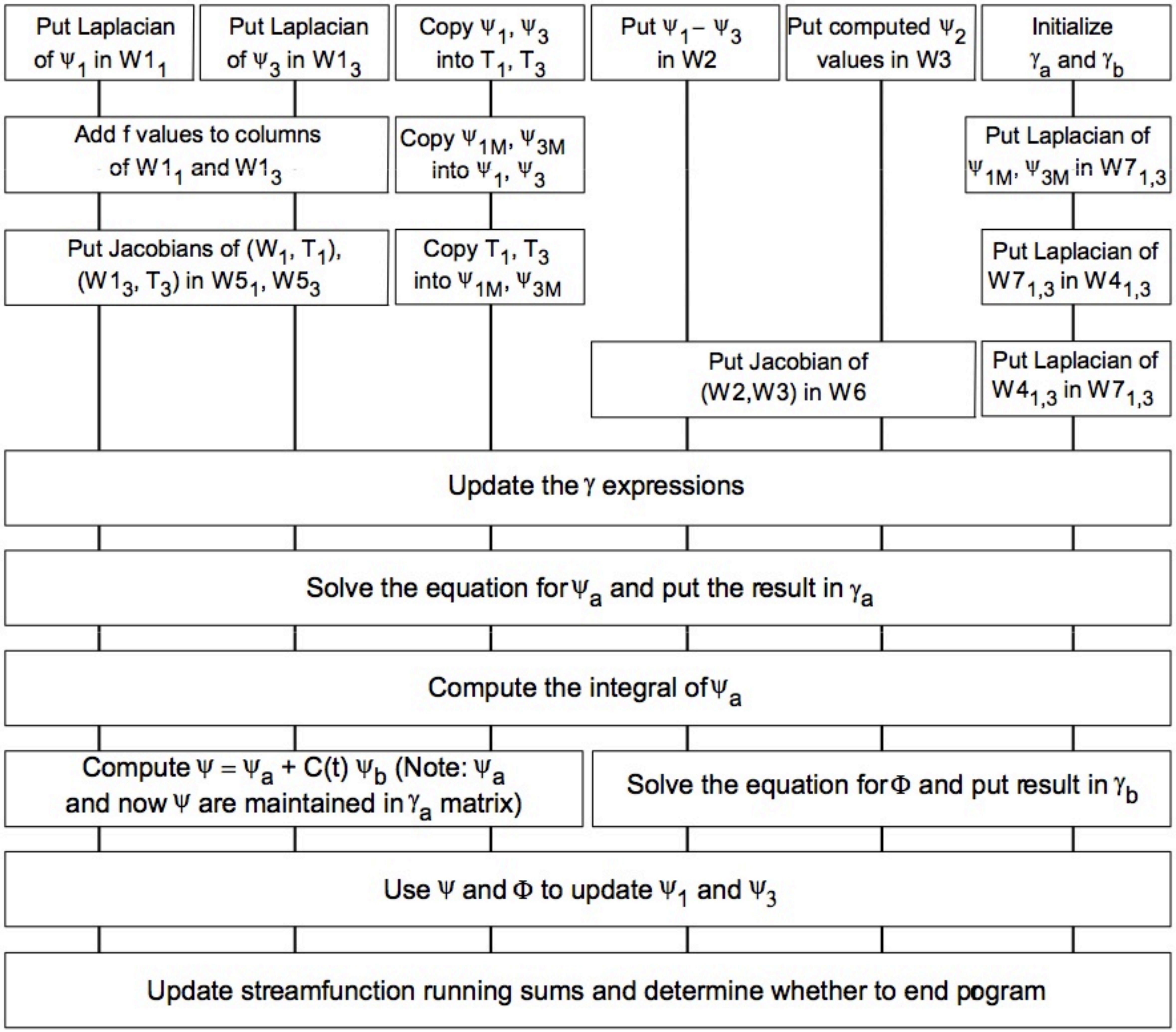
- Will be describing key aspects of the algorithms
  - Focus on: optimization techniques, analysis of workload characteristics

Case study 1: simulating ocean currents



- Discretize ocean into slices represented as 2D grids
- Discretize time evolution: Δt
- High accuracy simulation $\rightarrow$ small Δt and high resolution grids

Ocean: dependencies in one time step



Boxes correspond to computations on grids

Potential for parallelism within a grid (data-parallelism) and across operations on the different grids. Implementation only leverages data-parallelism. (simplicity)

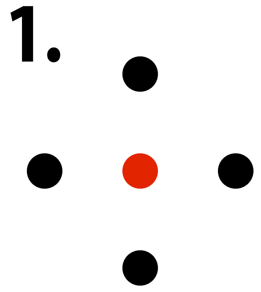
Ocean implementation details

- Static assignment: block decomposition (as previously discussed)

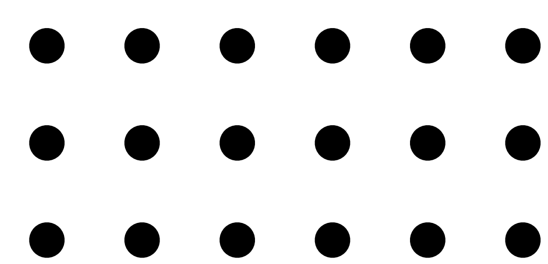
- Synchronization
  - Barriers (each grid computation is a phase)
  - Locks for mutual exclusion when updating shared variables (primarily for global reductions)

- Critical working sets
  1. Local neighborhood
  2. 3 rows of local partition
  3. Local partition

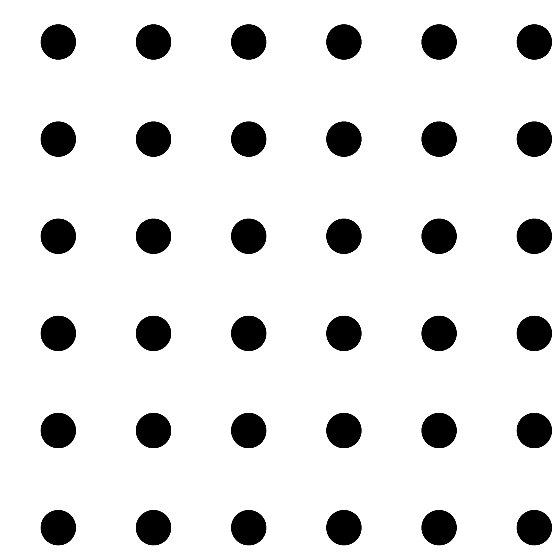
1.



2.

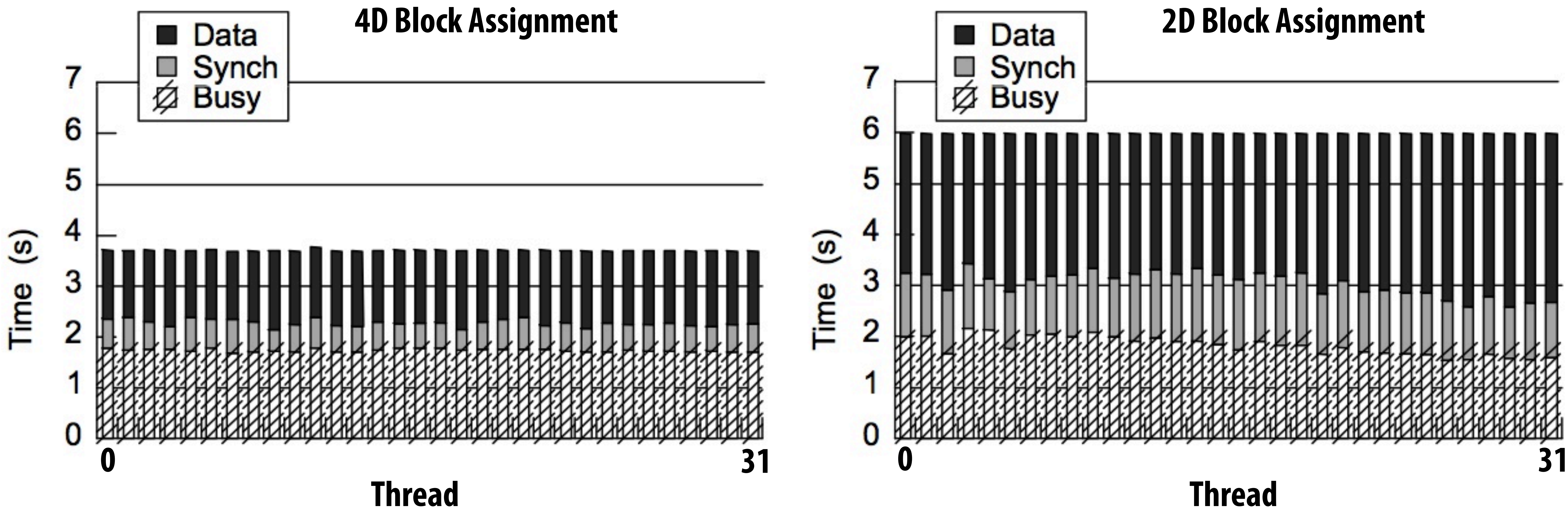


3.



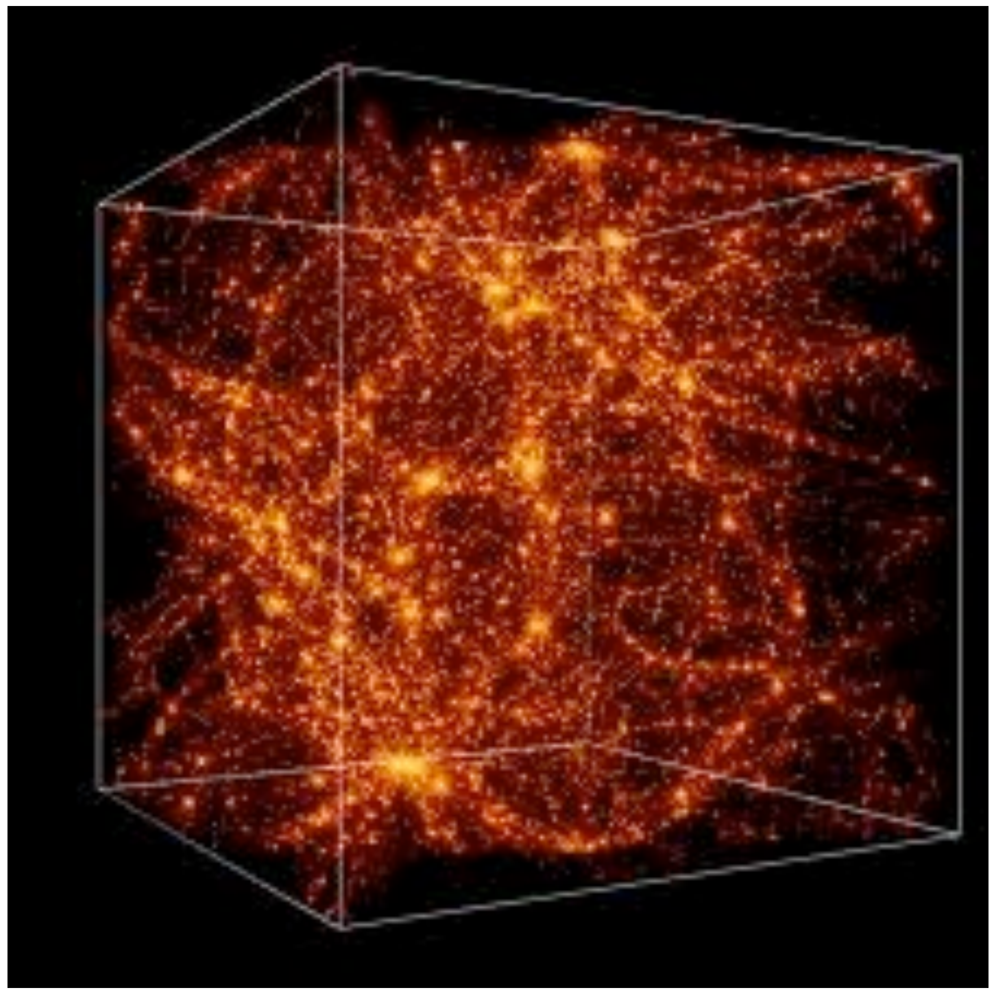
Ocean: execution time breakdown

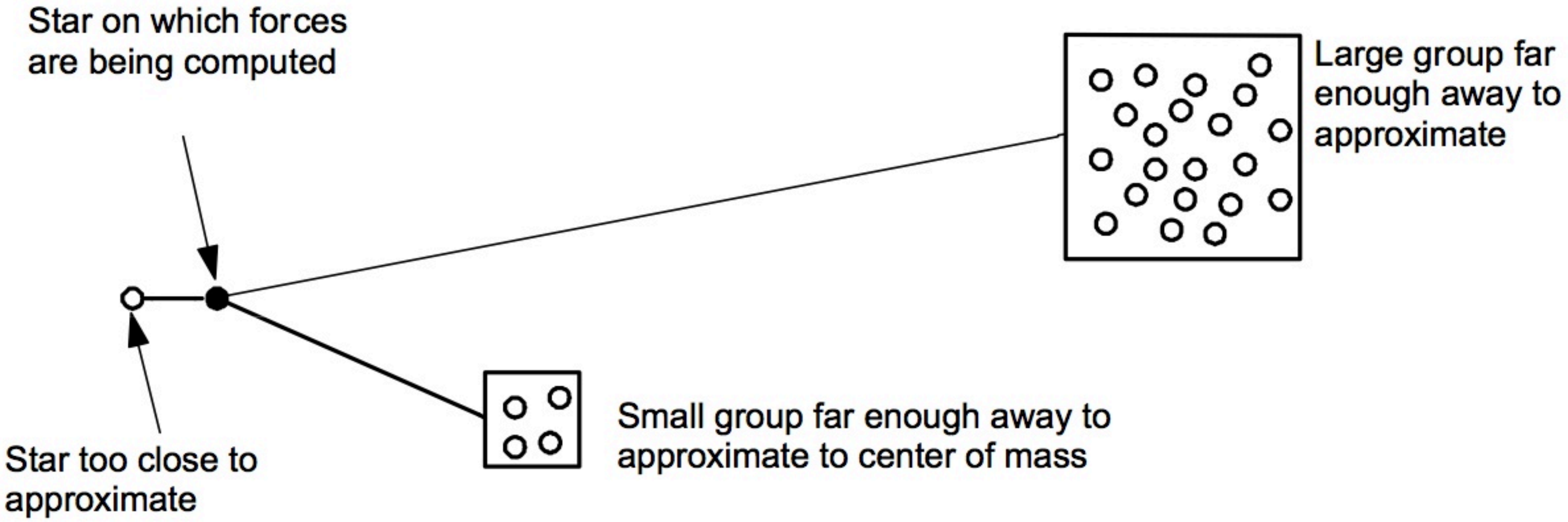
Execution on 32-processor SGI Origin 2000 (1026x1026 grids)



- Static assignment is sufficient (approx. equal busy time per thread)
- 4D blocking of grid reduces communication
(reflected on graph as data wait time)
- Synchronization cost largely due to waiting at barriers

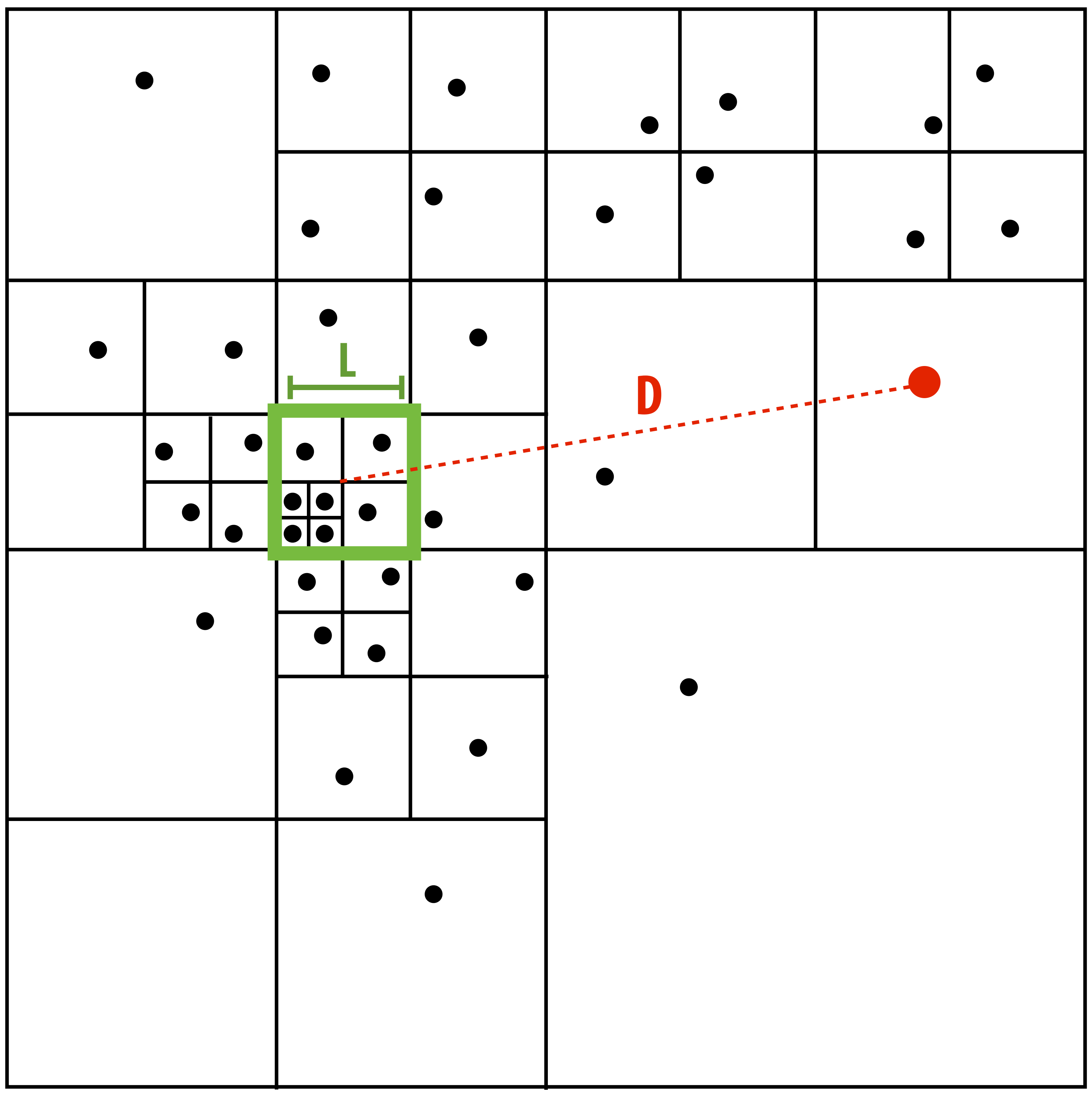
Case study 2: Galaxy evolution
Barnes-Hut algorithm



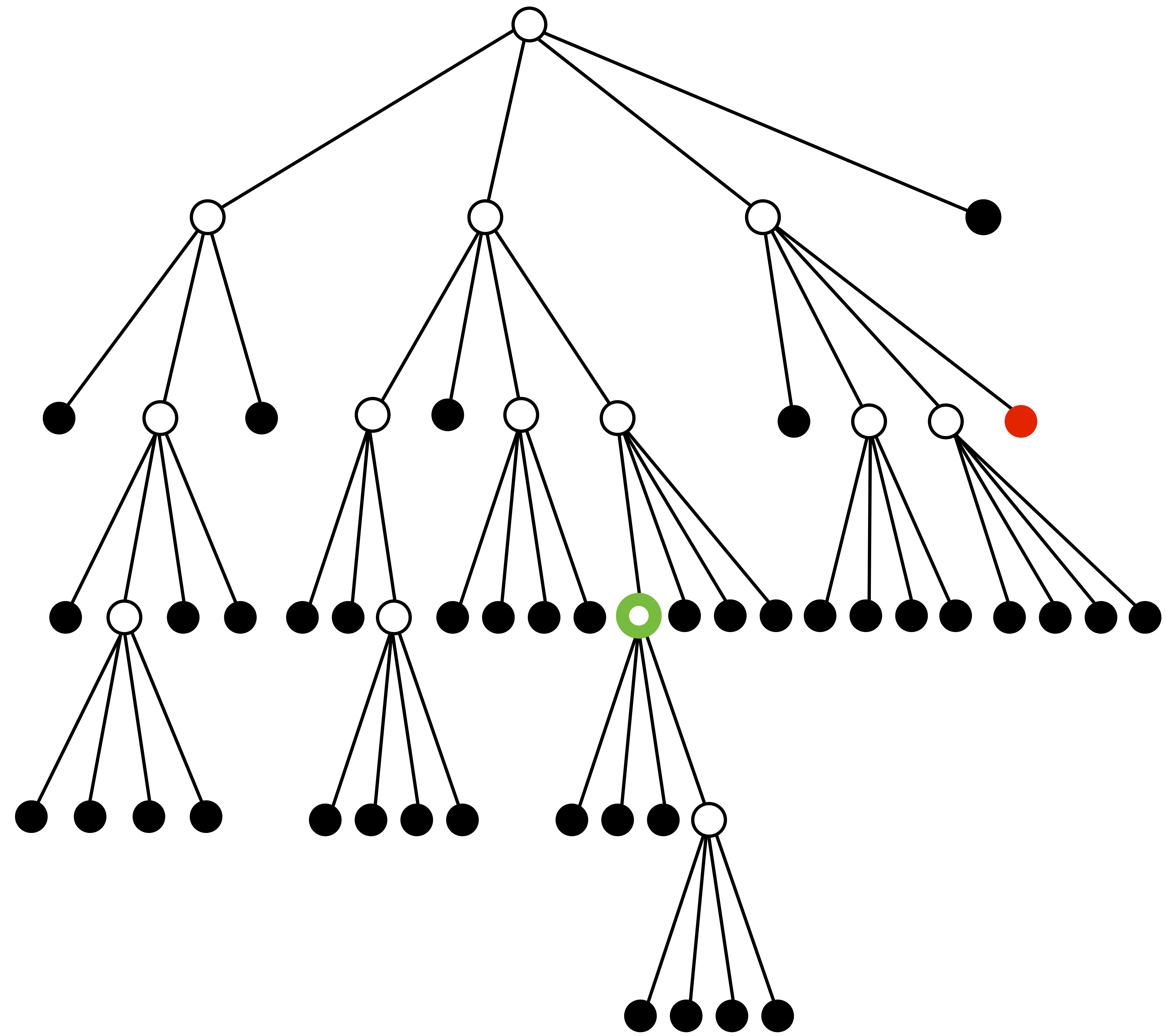


- Represent galaxy as a bunch of particles (think: particle = star)
- Compute forces due to gravity
  - Gravity has infinite extent: naive algorithm is $O(N^2)$
  - Magnitude of gravitational force falls off with distance (approximate forces from groups of far away stars)
  - Result is an O(NlgN) algorithm for computing gravitational forces between all stars

Barnes-Hut tree



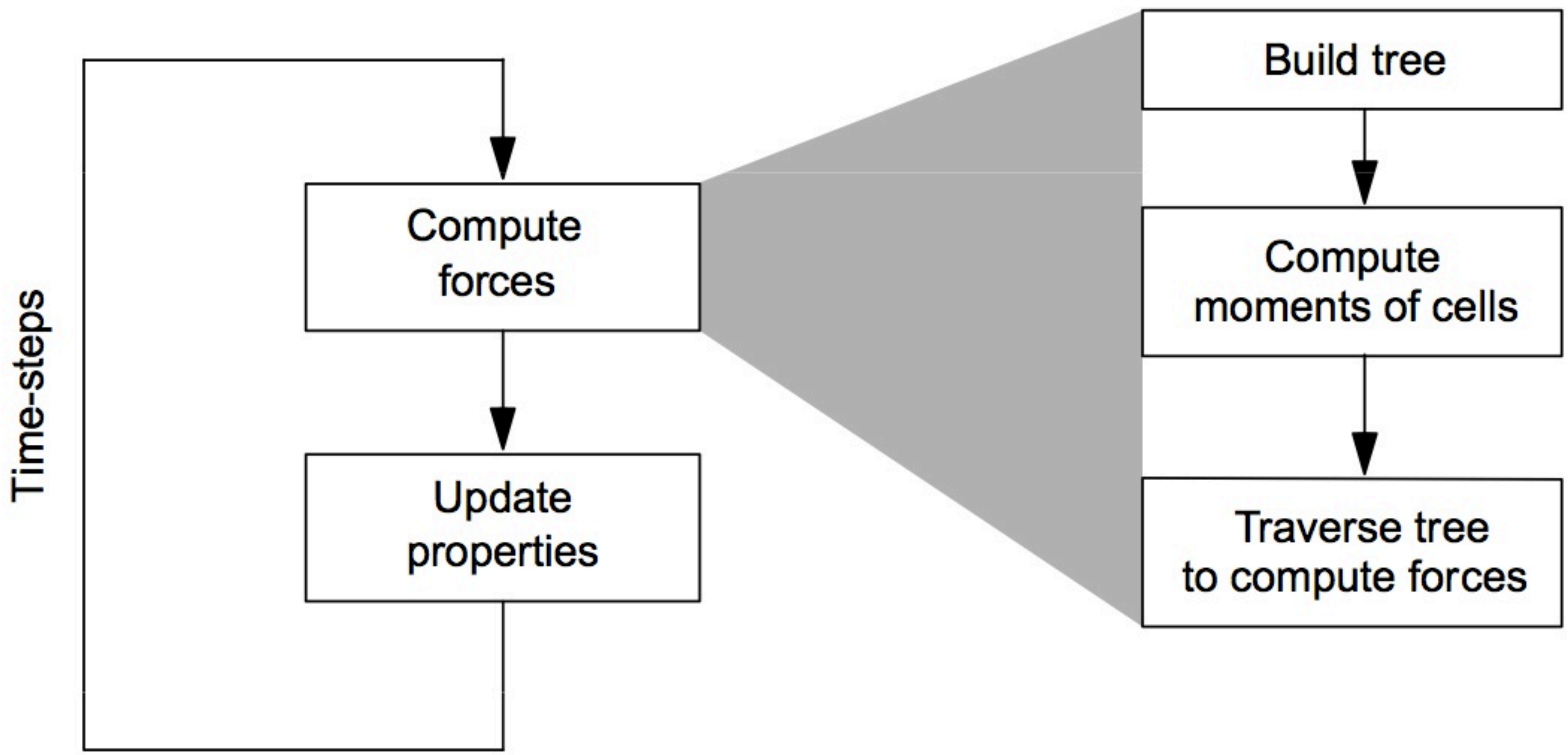
Spatial Domain



Quad-Tree Representation

- Interior nodes store center of mass, aggregate mass of child bodies (stars)
- For each body, traverse tree
- Compute forces using aggregate interior node if $L/D < \Theta$, else descend
- Expected number of nodes touched $O(\lg n / \Theta^2)$

Application structure



- **Challenges:**
  - **Amount of work per body, and communication pattern of work is non-uniform (depends on local density)**
  - **The bodies move: so costs and communication patterns change over time**
  - **Irregular, fine-grained computation**

- **But, there is a lot of locality in the computation (bodies near in space require similar data to compute forces -- should co-locate these computations!)**

Assignment

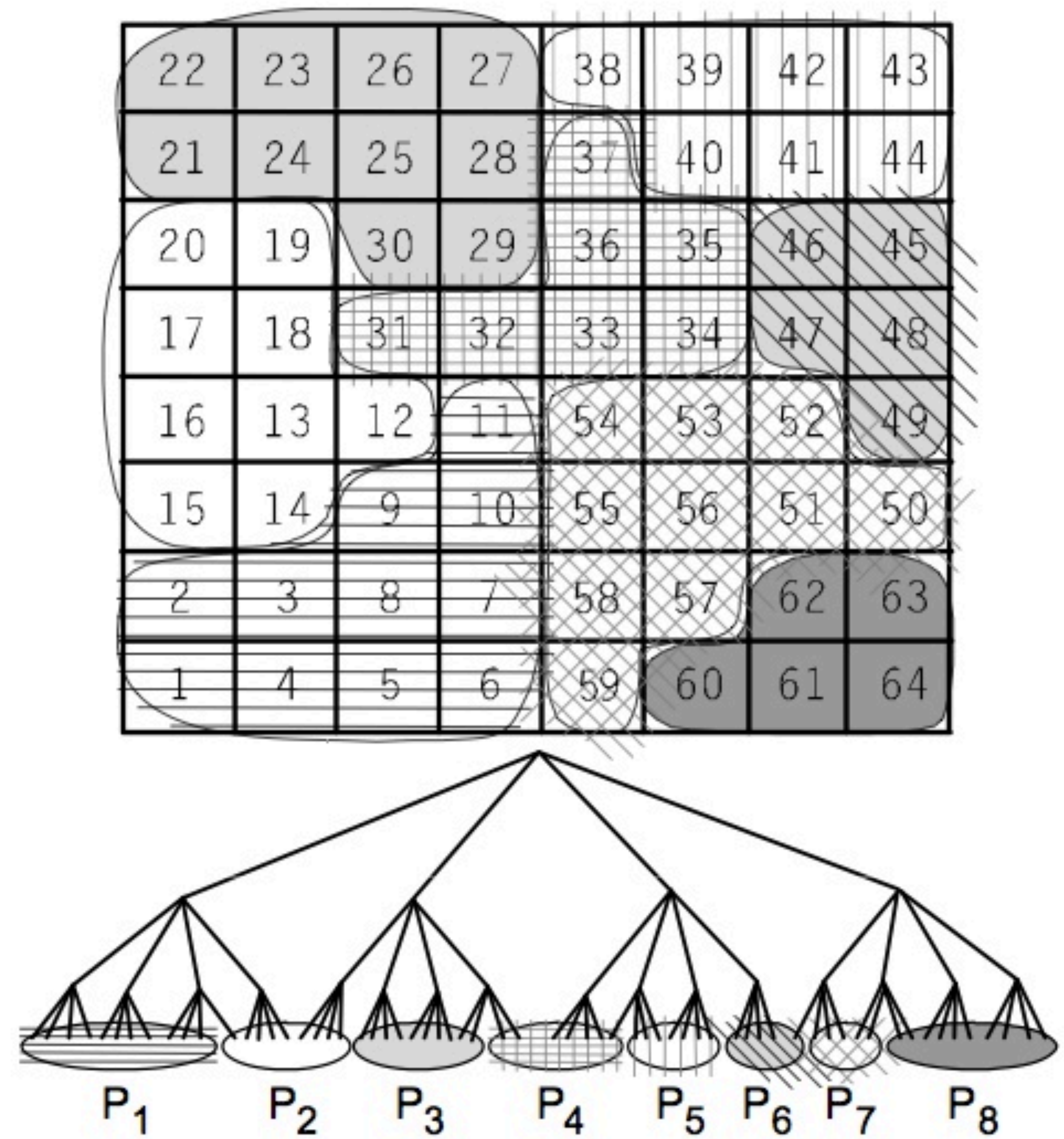
- Challenge:
  - Equal number of bodies per processor != equal work per processor
  - Want equal work per processor AND assignment should preserve locality

- Observation: spatial distribution of bodies evolves slowly

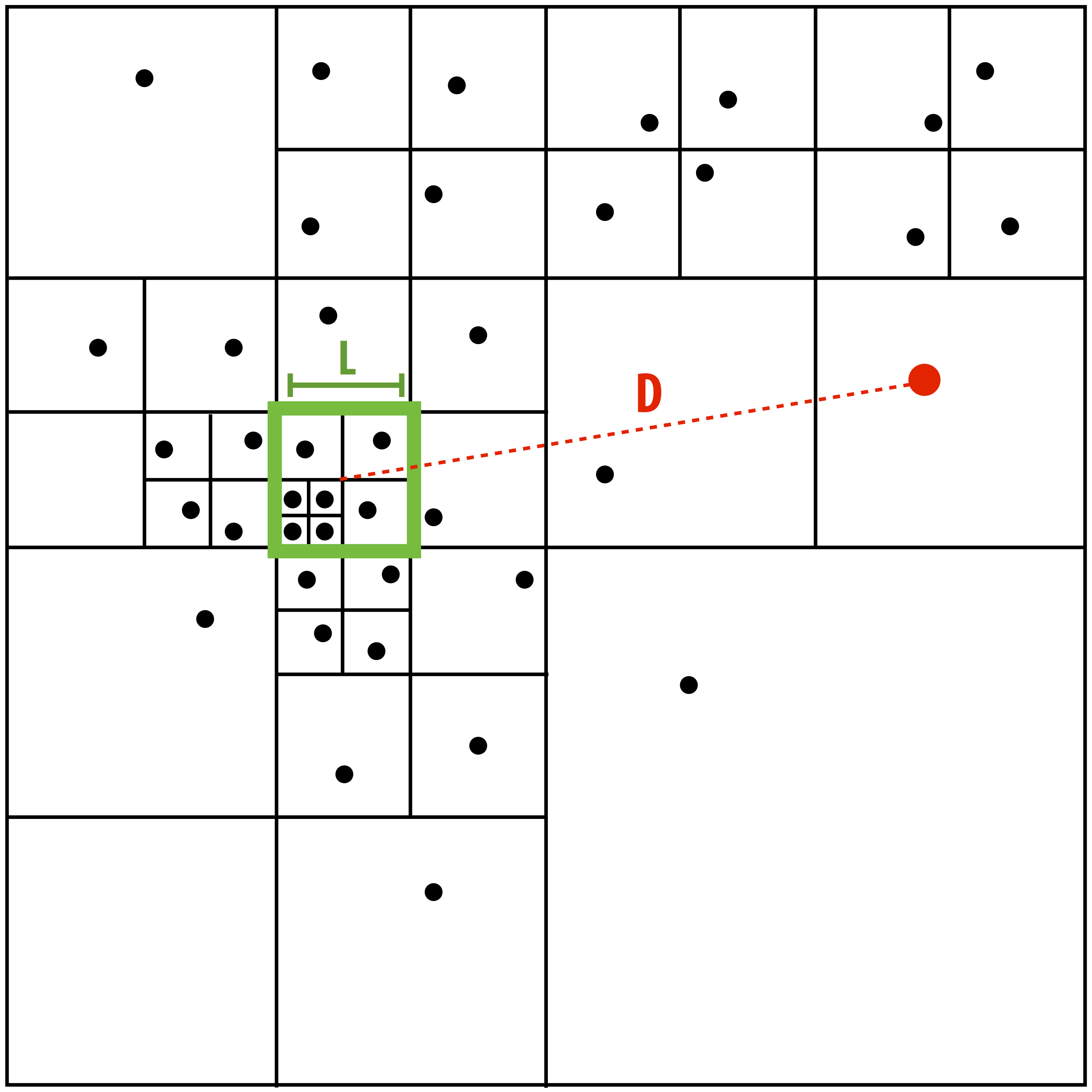
- Use semi-static assignment
  - Each time step, for each body, record number of interactions with other bodies (app self-profiles)
  - Cheap to compute. Just increment local counters
  - Use dynamic work costs to determine assignment

Assignment using cost zones

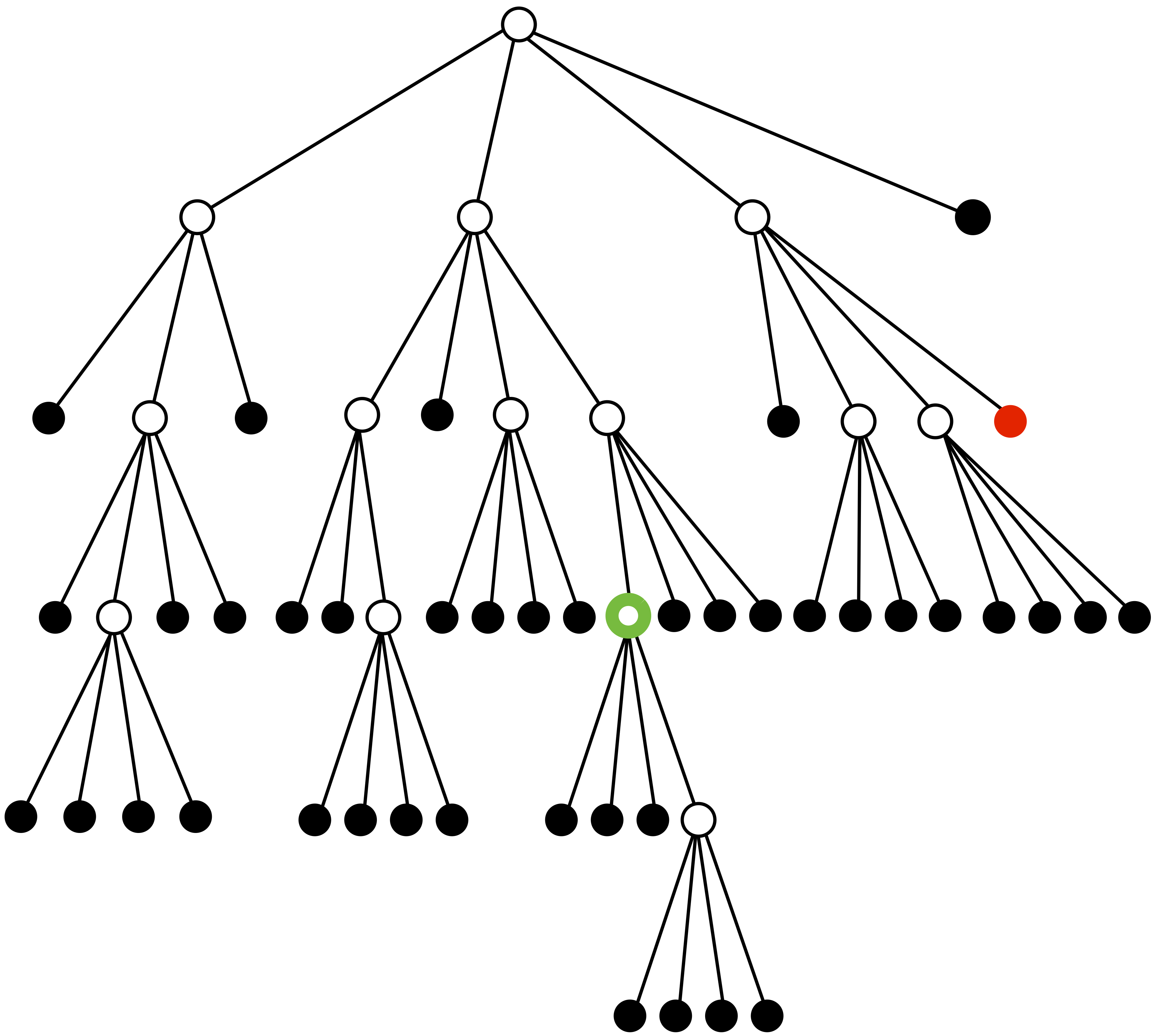
- Leverage locality inherent in tree
- Compute total work estimate W for all bodies (computed easily from per body costs)
- Each processor gets W/P of the work
- Thread on each processor runs breath-first search through tree (accumulates work seen so far)
- Processor P_i responsible for processing bodies corresponding to work: iW/P - $(i+1)W/P$
- Each processor can independently compute it's bodies. Only synchronization required is the reduction to compute total work



Barnes-Hut: working sets



Spatial Domain

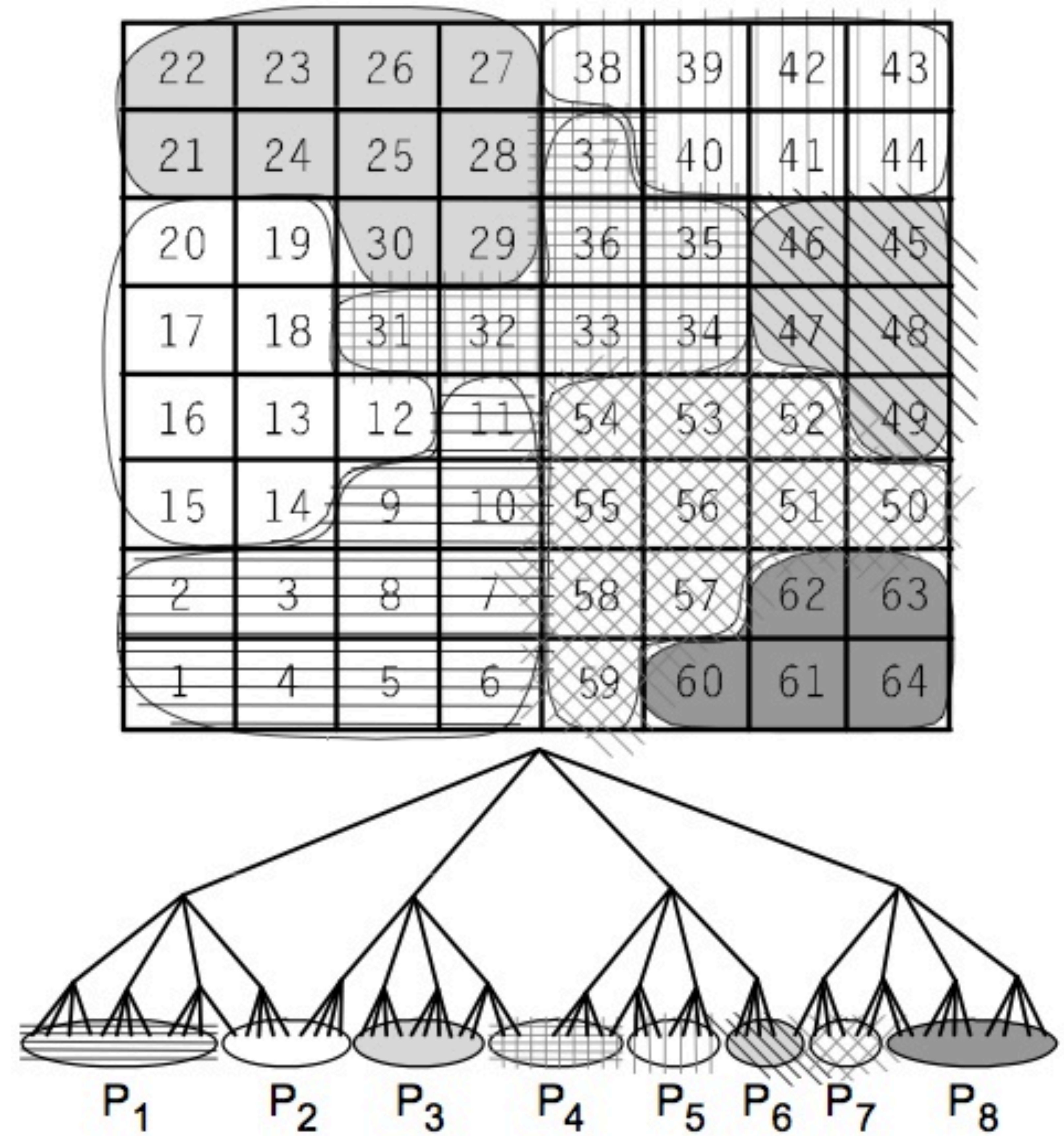


Quad-Tree Representation

- Working set 1: data needed to compute forces between body-body (or body-node) pairs
- Working set 2: data encountered in an entire tree traversal
  - Expected number of nodes touched for one body: $O(\lg n / \Theta^2)$
  - Next body is nearby, so it touches almost exactly the same nodes

Barnes-hut: data distribution

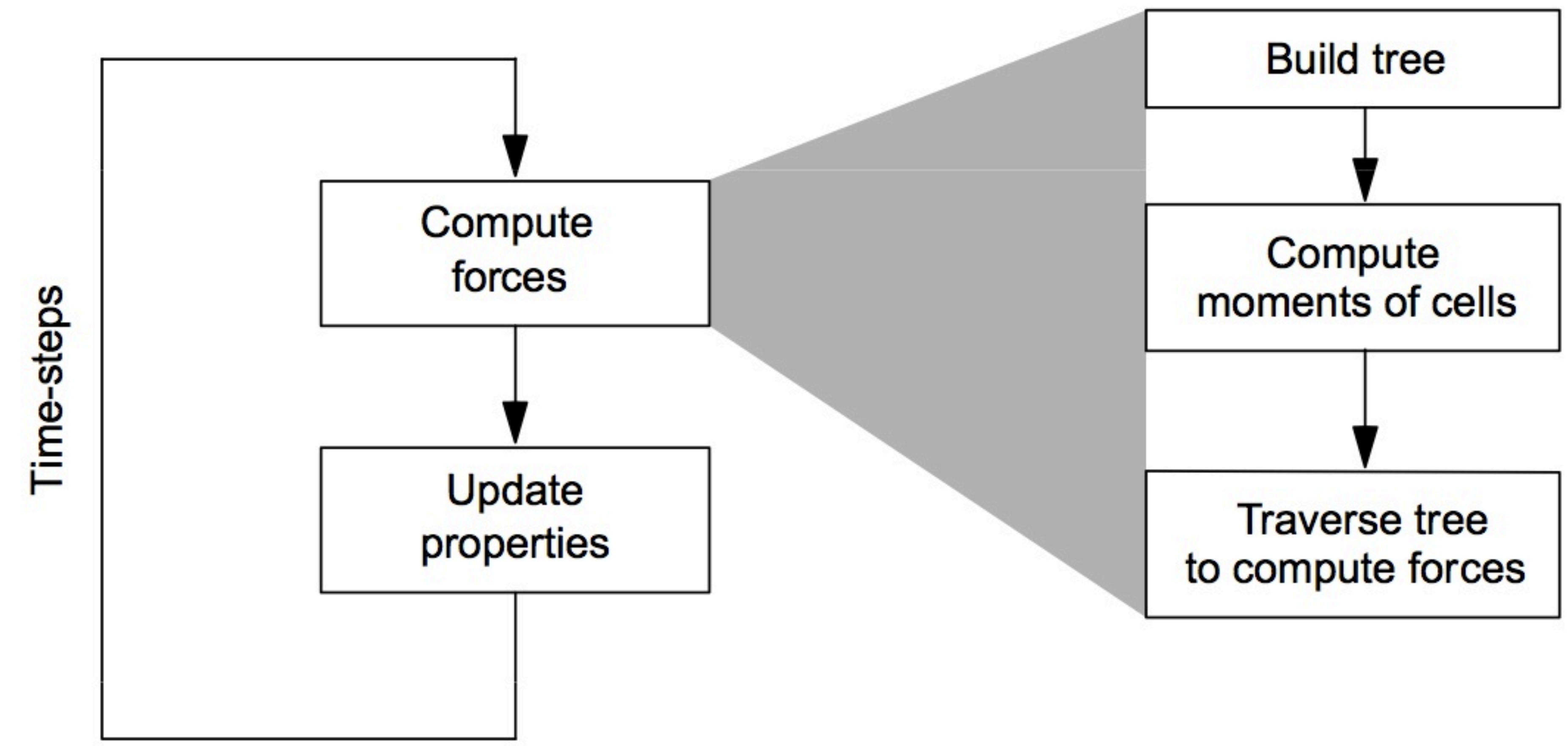
- Cost zones approach computes a work assignment. What about data distribution?
- Difficult to distribute data
  - Work assignment changes with time: would have to dynamically distribute data
  - Data accessed at fine granularity
- Luckily: large amounts of <u>temporal locality</u>
  - Bodies assigned to same processor are nearby space → tree nodes accessed during force computation are very similar.
  - Data just sits in cache (Barnes-Hut benefits from large caches, smaller cache line size)
- Result: Unlike OCEAN, data distribution in Barnes-Hut does not significantly impact performance
  - Use static distribution (interleaved) throughout the machine



Barnes-hut: synchronization

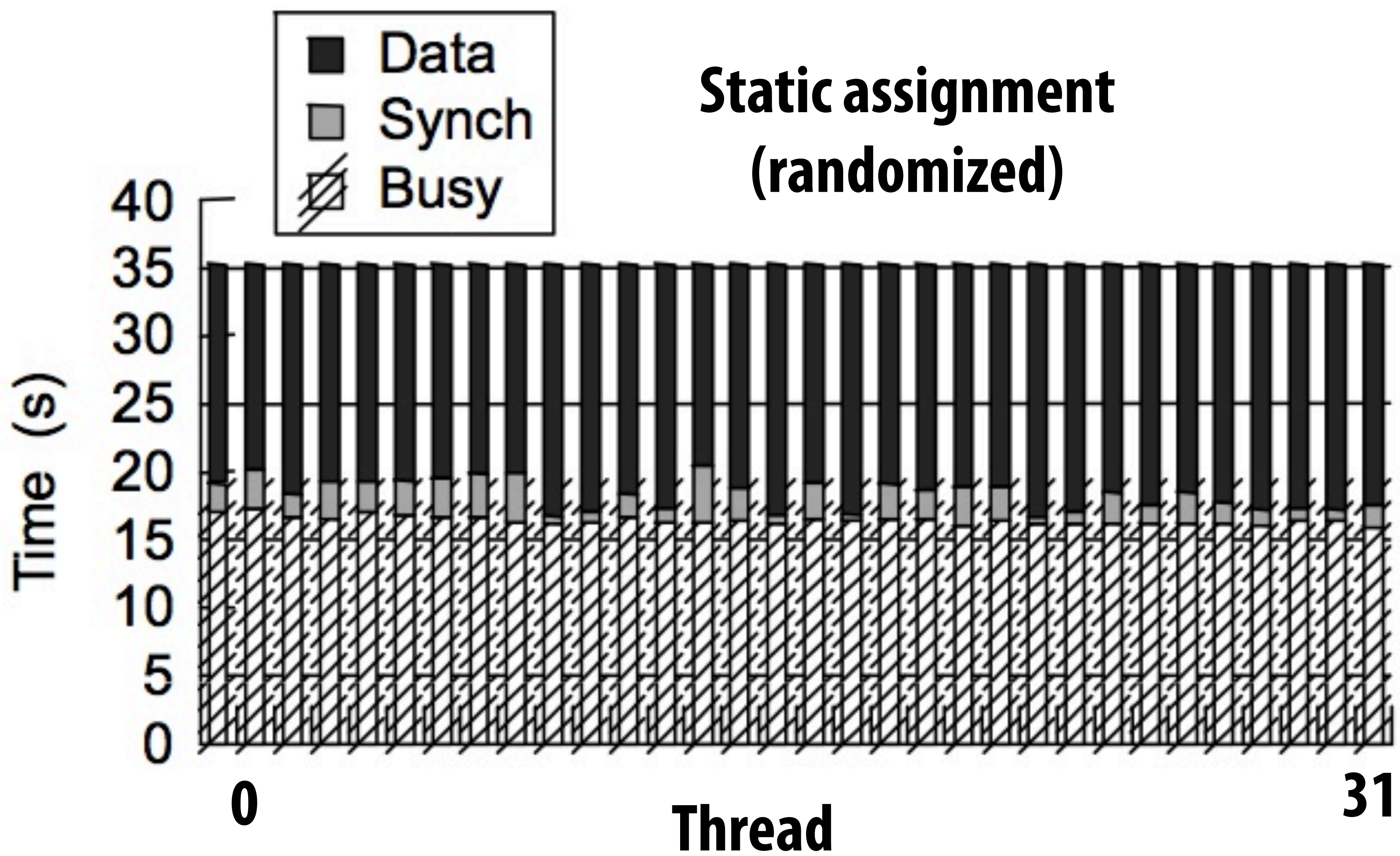
- **A few barriers between phases of force computation**

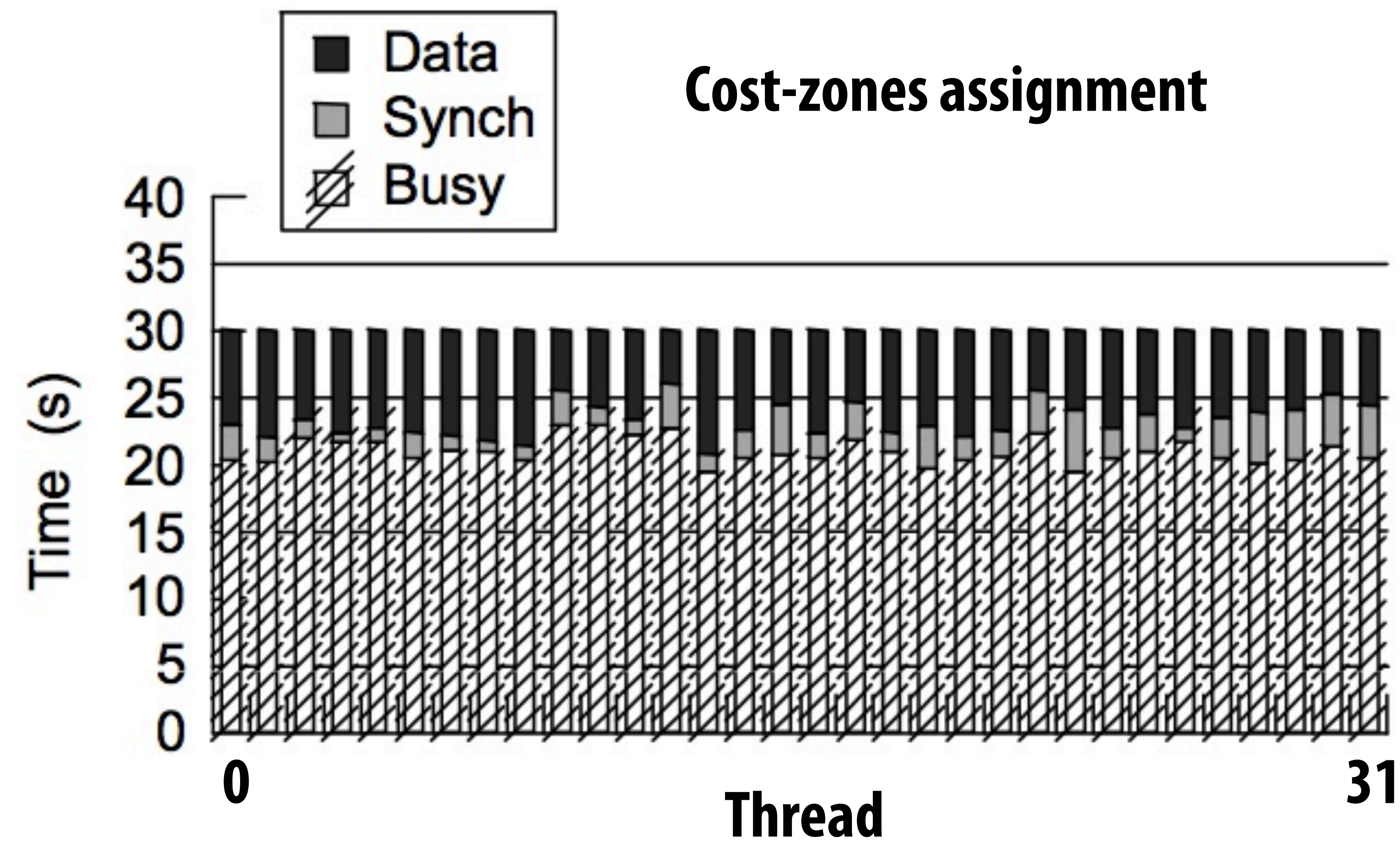
- **Fine-grained synchronization needed during tree build phase**
  - **Lock per tree cell**



Barnes-hut: execution time

Execution on 32-processor SGI Origin 2000 (512K bodies)





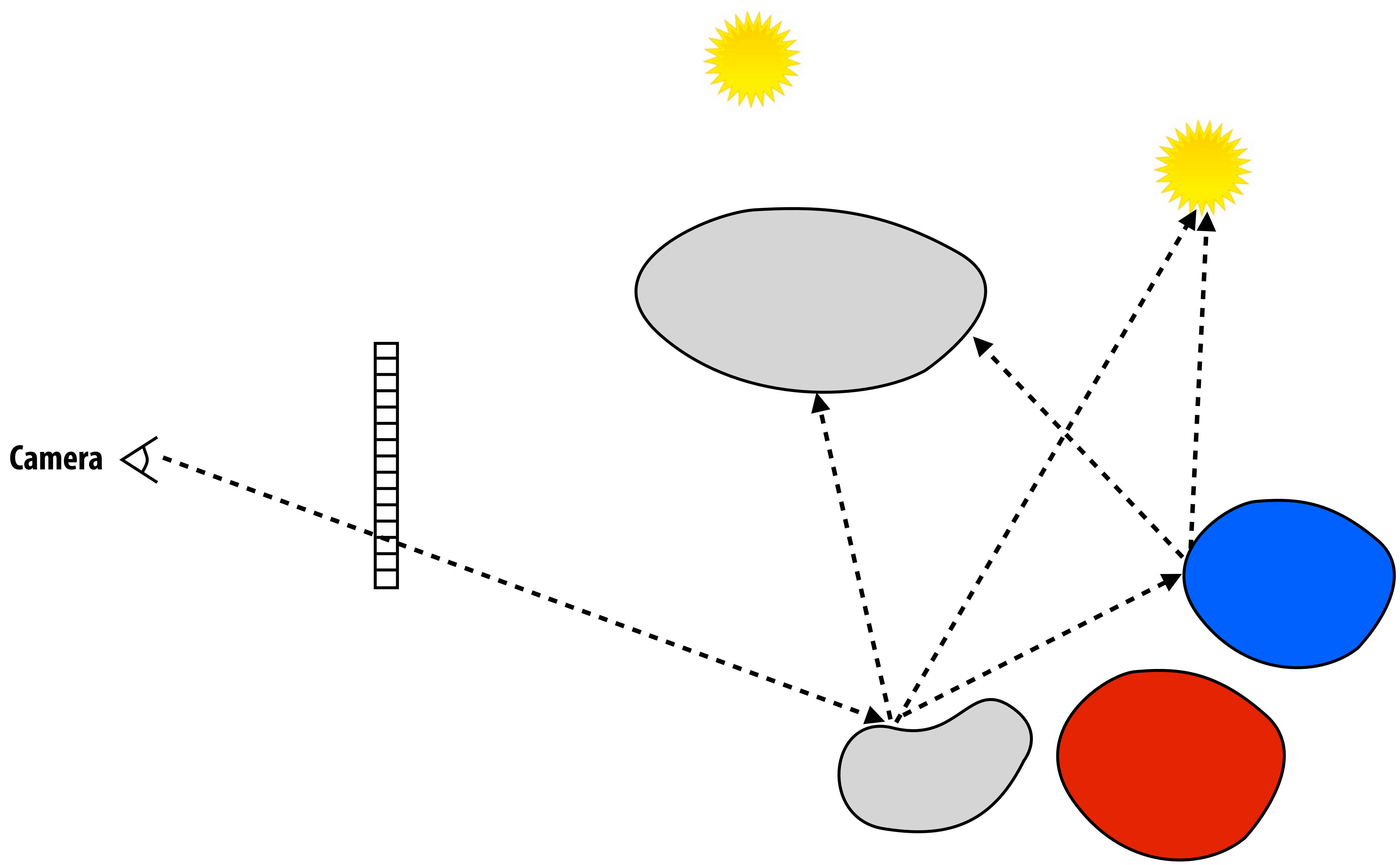
- **Load balance good even with static assignment because of random assignment**
  - **Law of averages: on average, each processor does approx. the same amount of work**

- **But random assignment yields poor locality**
  - **Significant amount of inherent communication**
  - **Significant amount of artifactual communication (fine-grained accesses)**

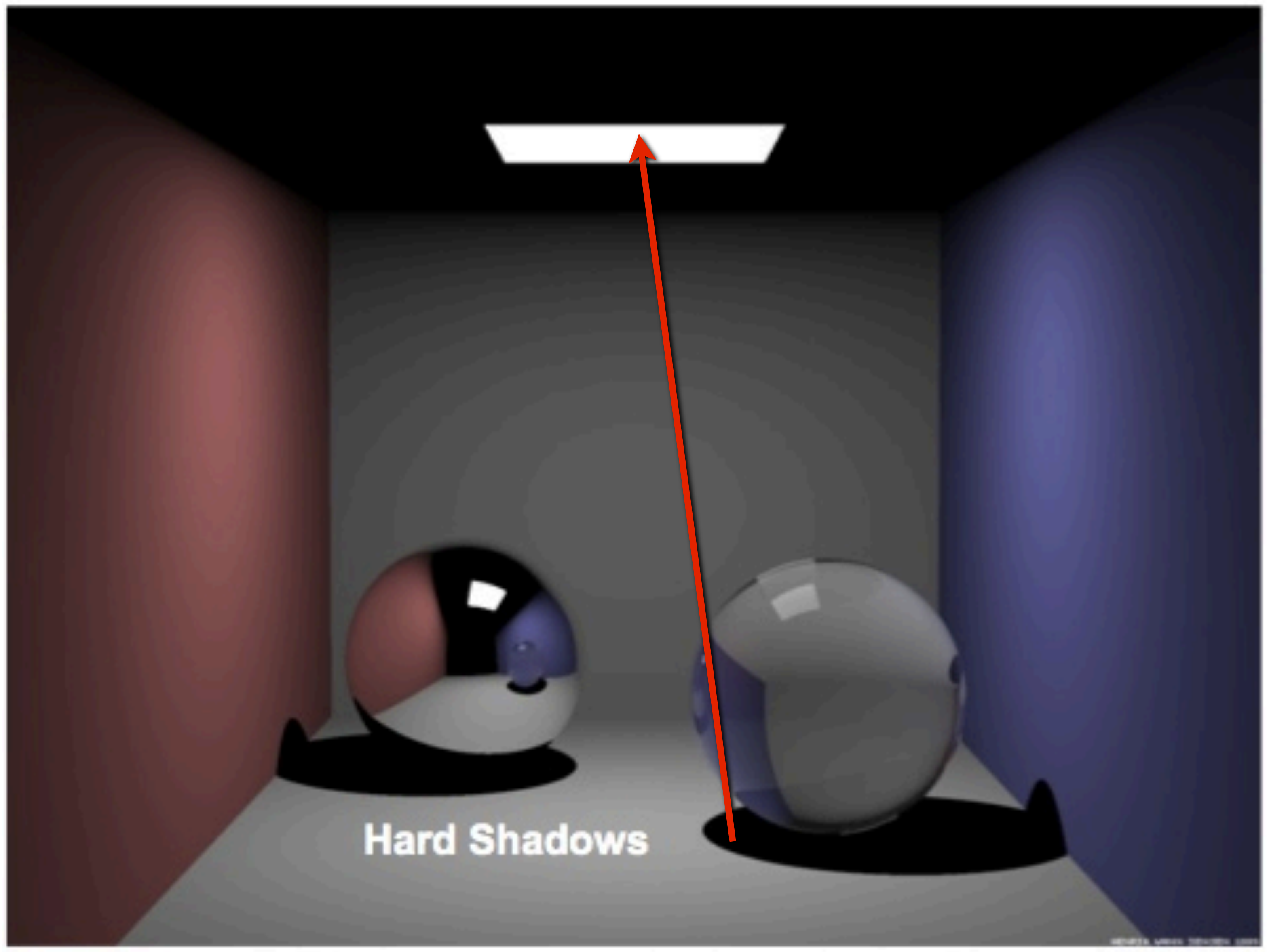
- **Common tension: work balance vs. locality (cost-zones get us both!)**
  (similar to work balance vs. synchronization trade-offs in previous lecture)

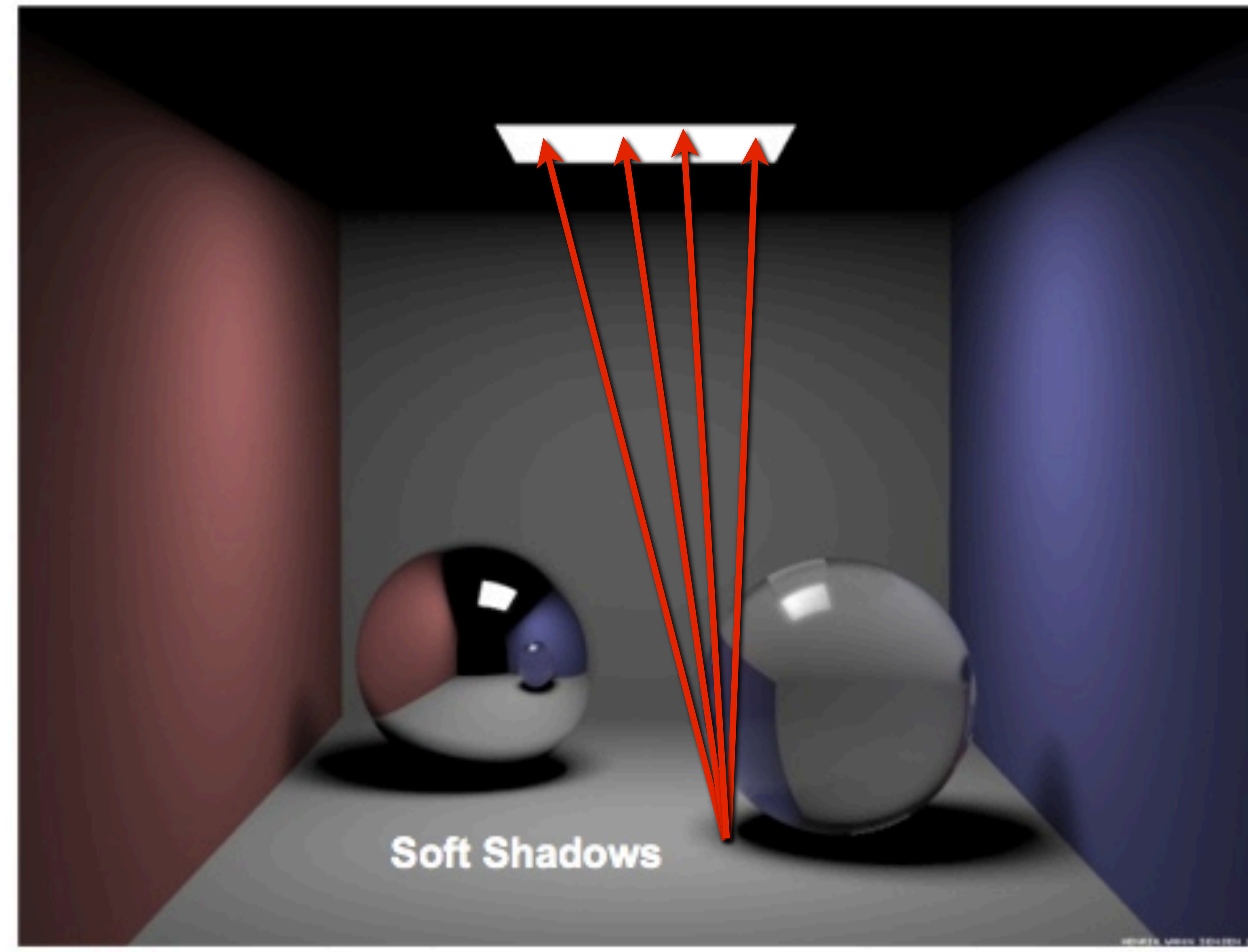
Case study 3: ray tracing

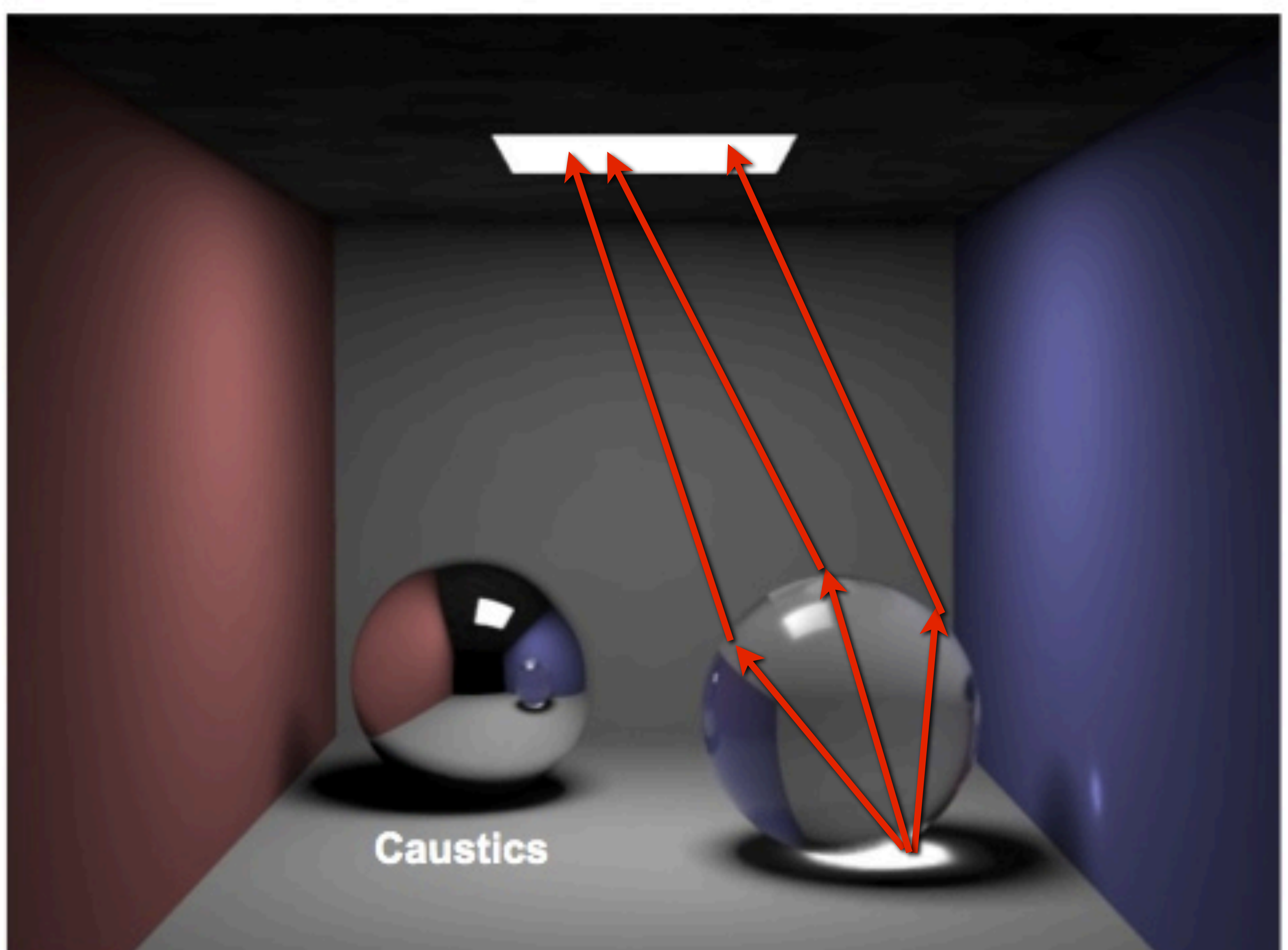
- **Synthesize images of a complex scene**
  - What scene geometry is intersected by each ray?
  - Which intersection is closest?
  - How much light reaches the camera from this surface point



Sampling light paths







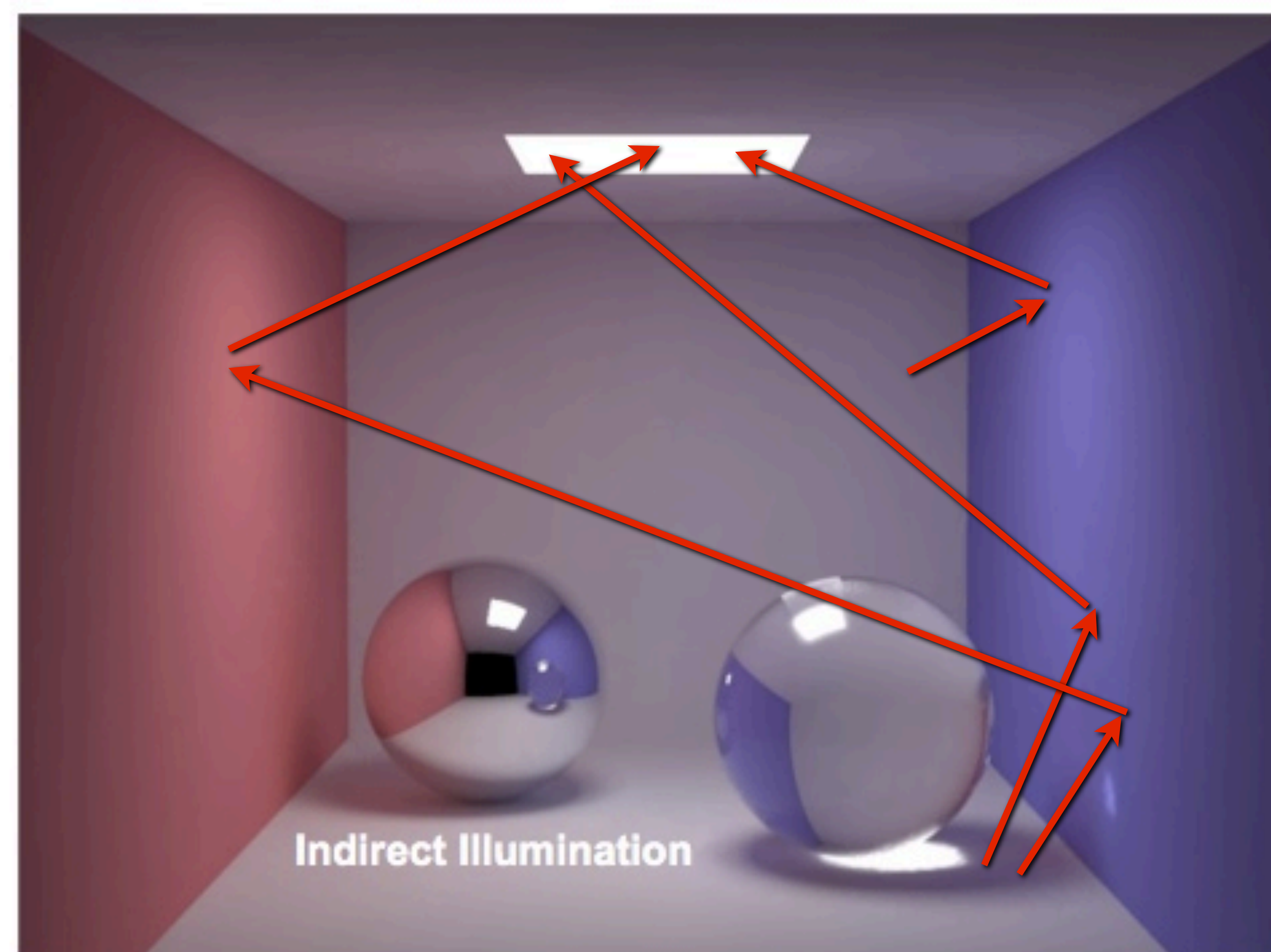
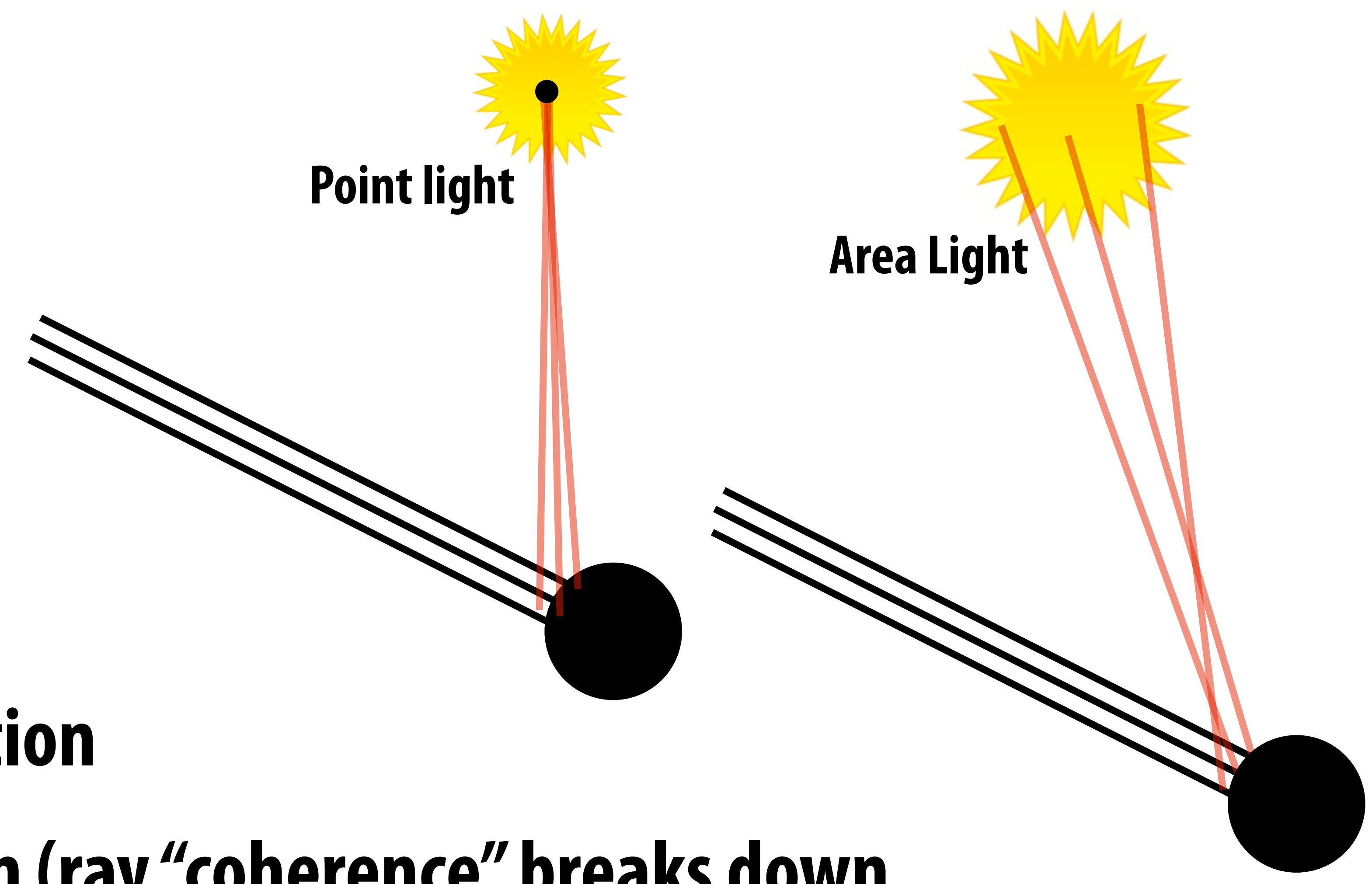


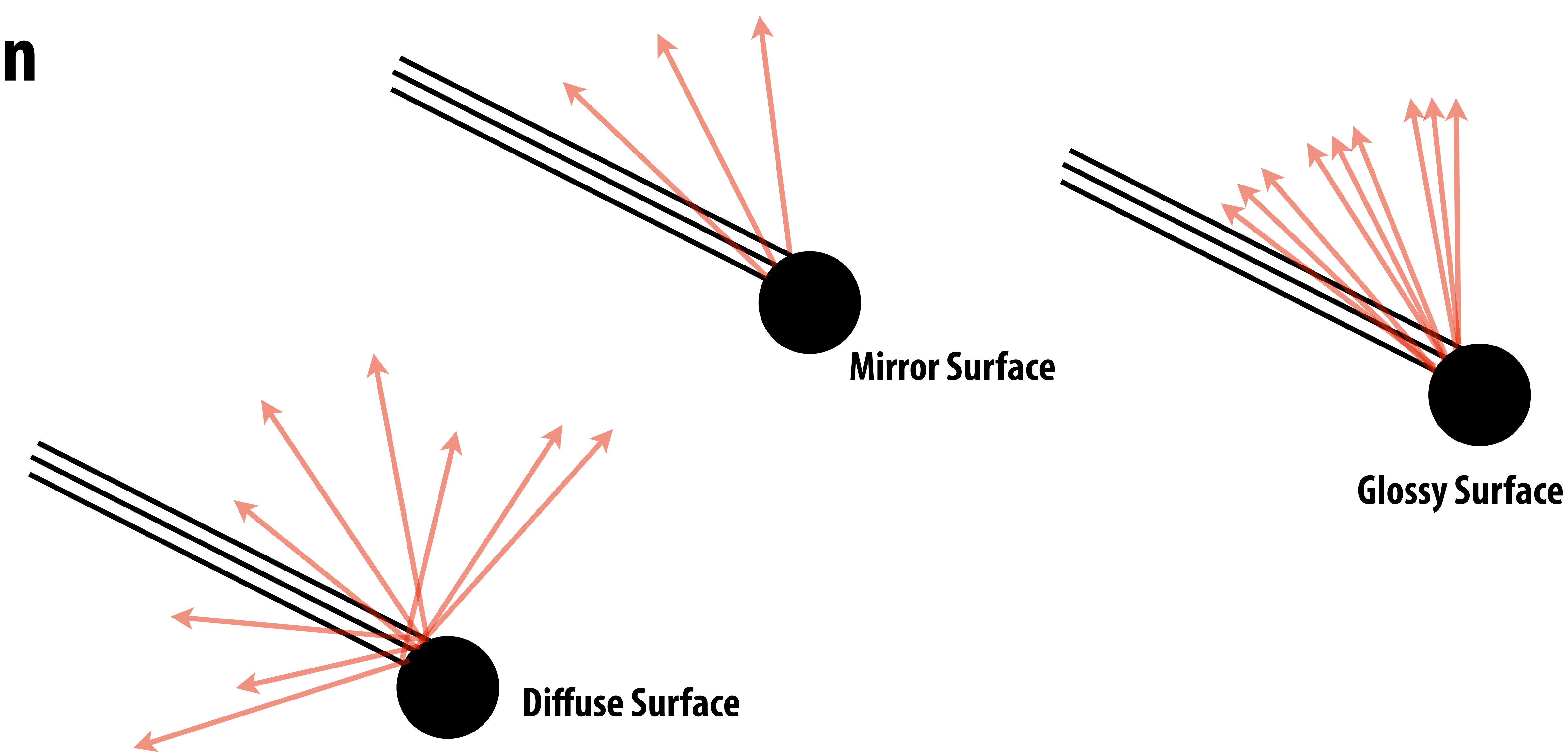
Image credit: Wann Jensen, Hanrahan

Types of rays

- **Camera (a.k.a., eye rays, primary rays)**
  - Common origin, similar direction
- **Shadow**
  - Point source: common destination, similar direction
  - Area source: similar destination, similar direction (ray "coherence" breaks down as light source increases in size: e.g., consider entire sky as an area light source)

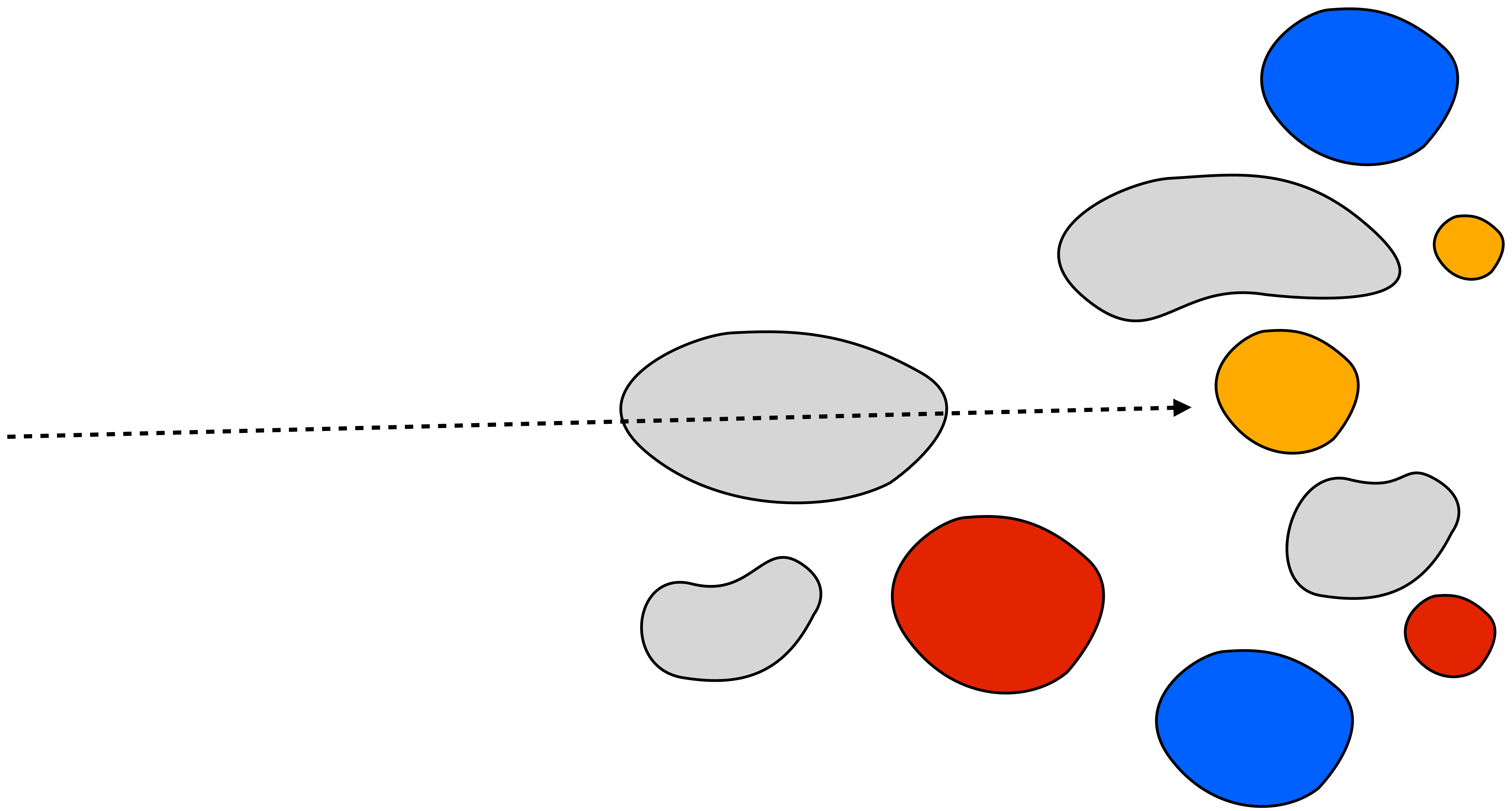


- **Indirect illumination**
  - Mirror surface
  - Glossy surface
  - Diffuse surface



Problem

Given ray, find first intersection with scene geometry **



** Another common query: determine if <u>any</u> intersection exists

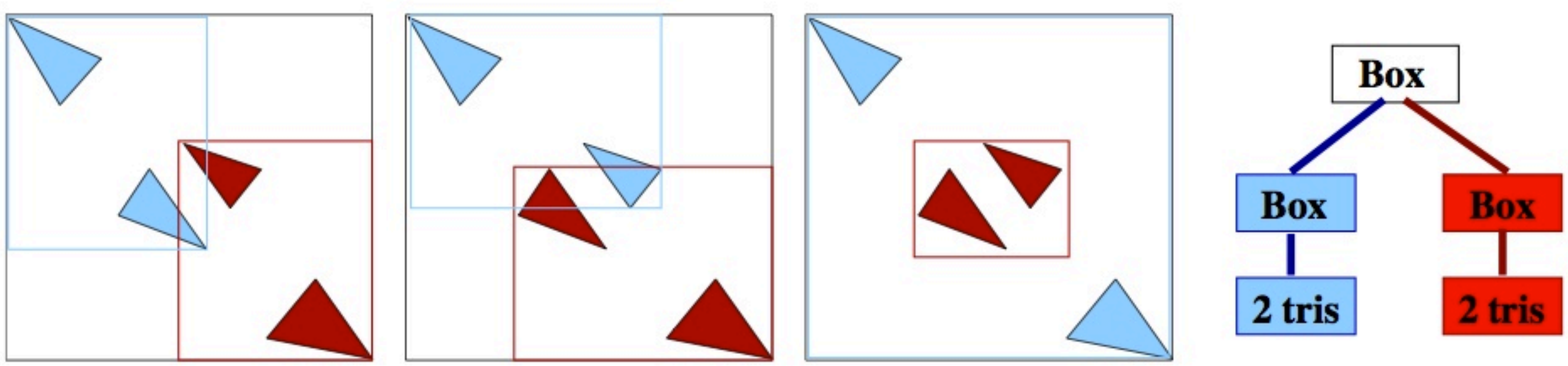
Acceleration structures

Preprocess scene to build data structure to accelerate ray-scene visibility queries

e.g., bounding volume hierarchy (BVH)

Idea: nodes group objects with spatial proximity (like quad-tree in Barnes-hut)

Adapts to non-uniform density of scene objects



Three different bounding volume hierarchies for the same scene

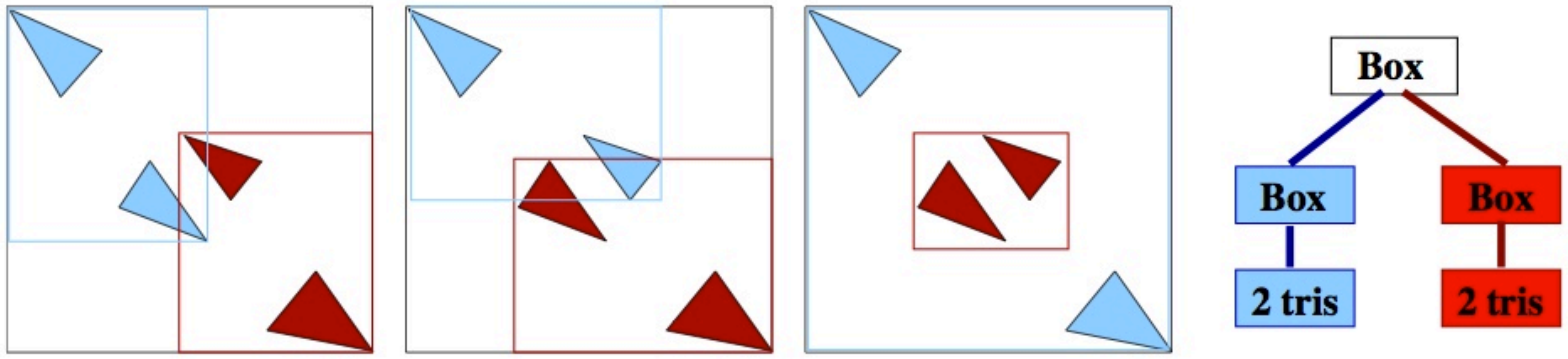
Image credit: Wald et al. TOG 2004

High-throughput ray tracing

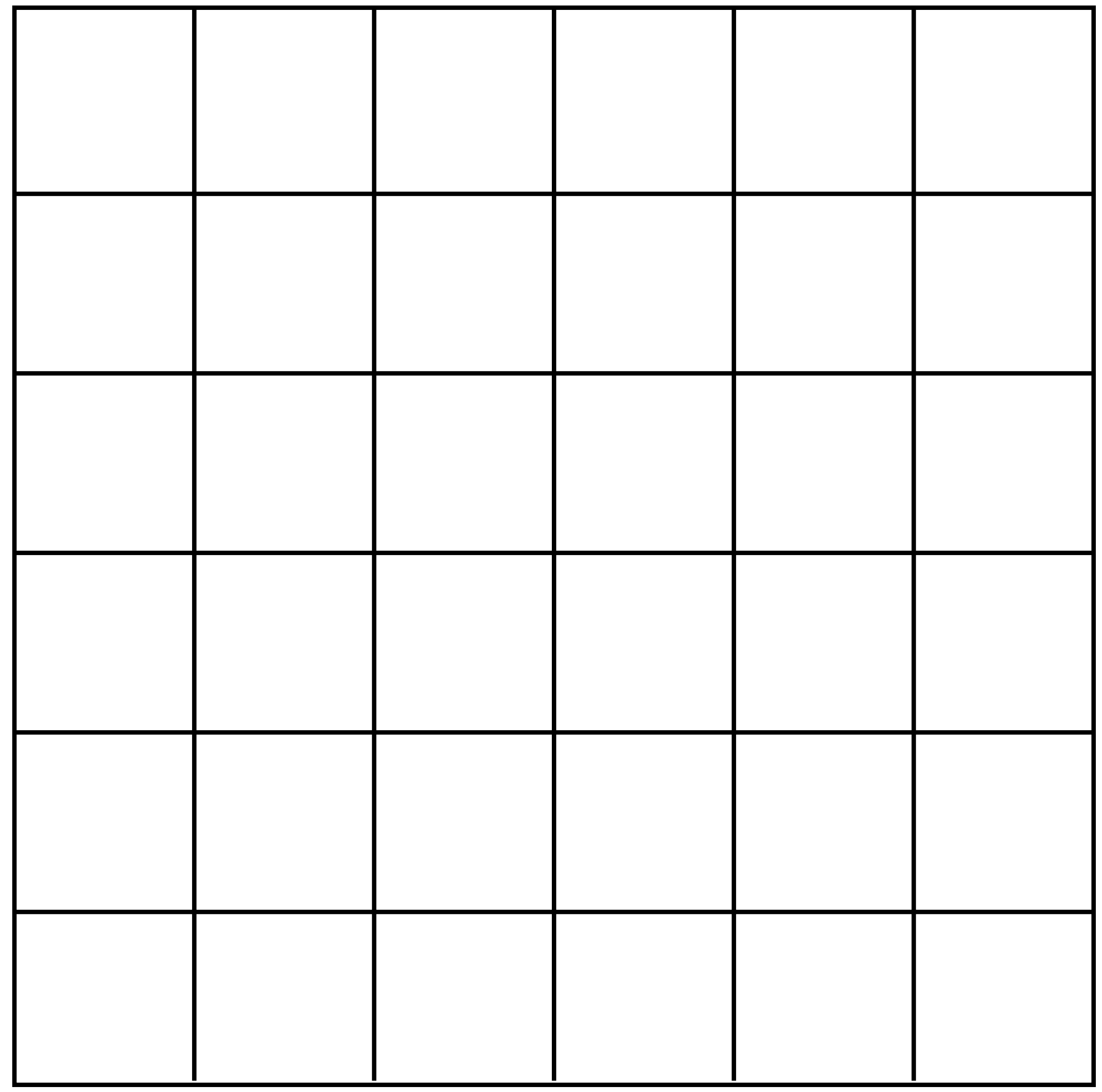
- **Work efficiency of algorithms**
  - High quality acceleration structures (minimize ray-box, ray-primitive tests)
  - Smart traversal algorithms (early termination, etc.)
- **Parallelism: multi-core, SIMD execution efficiency**
- **Bandwidth efficiency (caching, memory access characteristics)**

Simple ray tracer (using BVH)

```
// stores information about closest hit found so far
struct ClosestHitInfo {
   Primitive primitive;
   float distance;
};

trace(Ray ray, BVHNode node, ClosestHitInfo hitInfo)
{
   if (!intersect(ray, node.bbox) || (closest point on box is farther than hitInfo.distance))
      return;

   if (node.leaf) {
      for (each primitive in node) {
         (hit, distance) = intersect(ray, primitive);
         if (hit && distance < hitInfo.distance) {
            hitInfo.primitive = primitive;
            hitInfo.distance = distance;
         }
      }
   } else {
     trace(ray, node.leftChild, hitInfo);
     trace(ray, node.rightChild, hitInfo);
   }
}
```



Decomposition and assignment



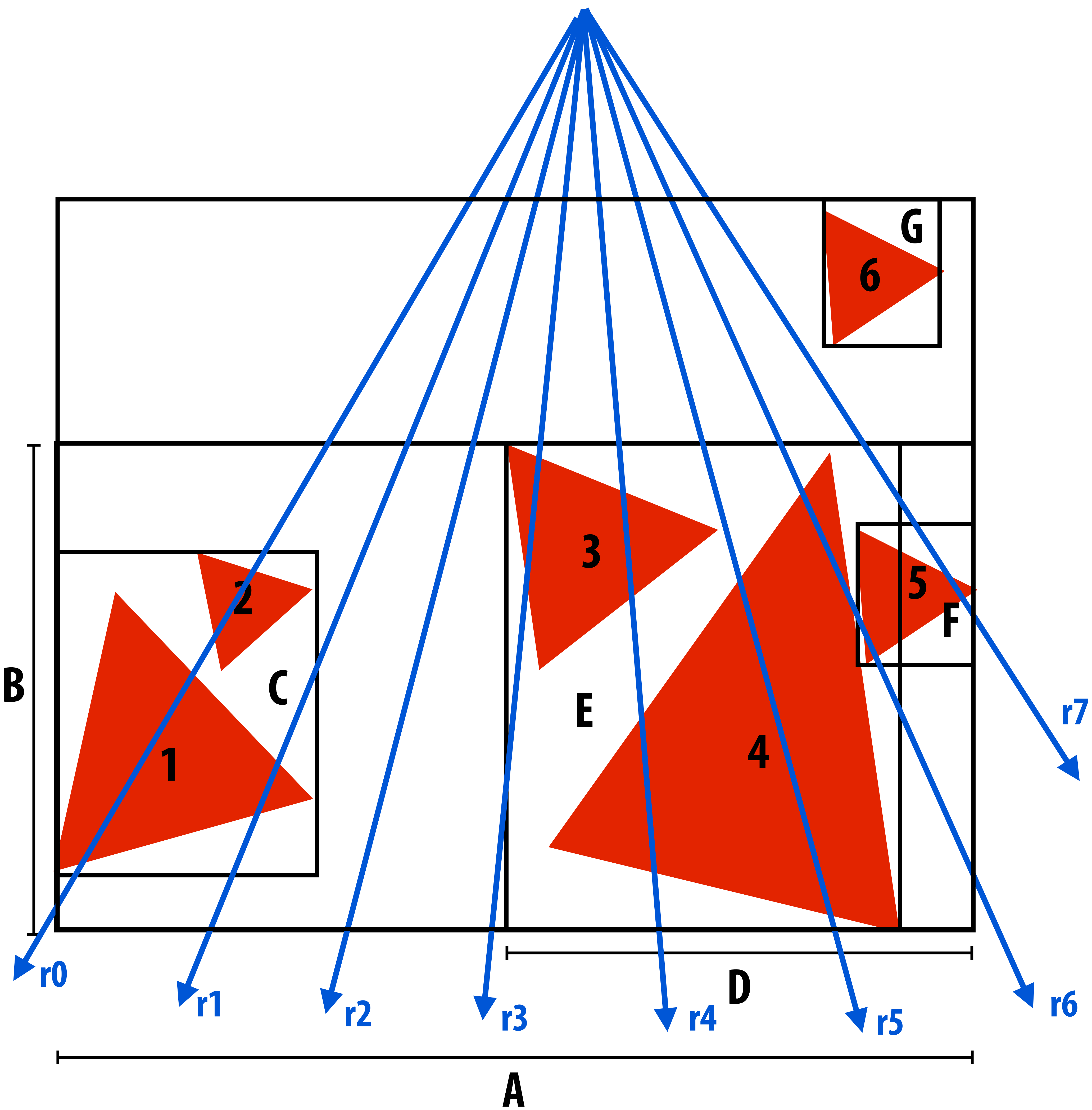
- Spatial decomposition of image (2D blocks)
  - 2D blocks maximize spatial locality of rays
- Create many more tiles than processors (just like assignment 1, problem 2)
- Use simple work queue to dynamically assign work to processors
  - Cost to render a block is large → synchronization cost trivial

Ray packet tracing

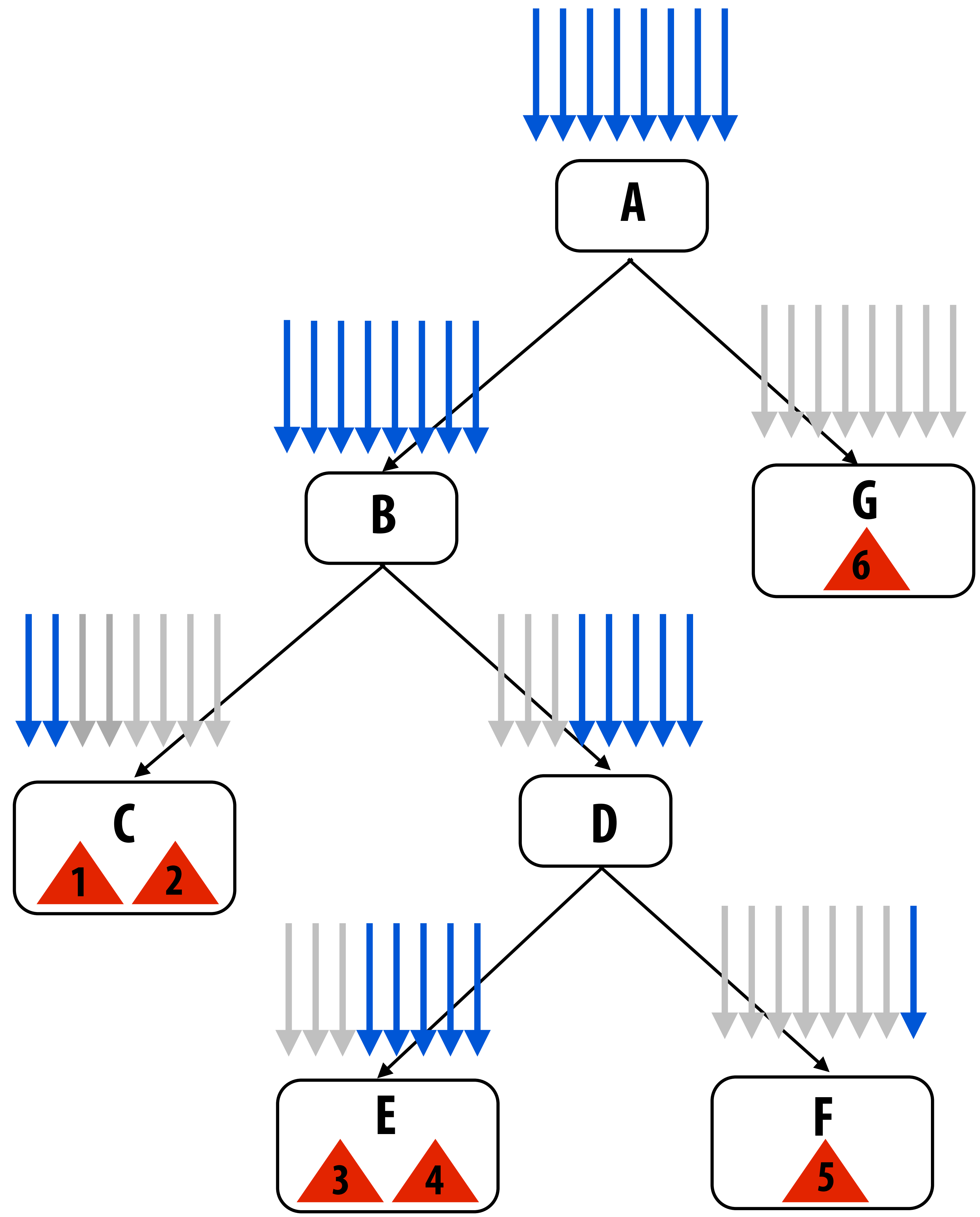
Program explicitly intersects a collection of rays against BVH at once

```
RayPacket
{
    Ray rays[PACKET_SIZE];
    bool active[PACKET_SIZE];
};

trace(RayPacket rays, BVHNode node, ClosestHitInfo packetHitInfo)
{
   if (!ANY_ACTIVE_intersect(rays, node.bbox) ||
       (closest point on box (for all active rays) is farther than hitInfo.distance))
      return;

   update packet active mask

   if (node.leaf) {
      for (each primitive in node) {
         for (each ACTIVE ray r in packet) {
            (hit, distance) = intersect(ray, primitive);
            if (hit && distance < hitInfo.distance) {
               hitInfo[r].primitive = primitive;
               hitInfo[r].distance = distance;
            }
         }
      }
   } else {
     trace(rays, node.leftChild, hitInfo);
     trace(rays, node.rightChild, hitInfo);
   }
}
```

Ray packet tracing



Blue = active ray after node box test



r6 does not pass node F box test due to closest-so-far check

Advantages of packets

- **SIMD execution**
  - One vector lane per ray
- **Amortize fetch: all rays in packet visit node at same time**
  - Load BVH node once for all rays in packet
  - Note: value to making packets much bigger than SIMD width!
  - Contrast with SPMD approach
- **Amortize work**
  - Packets are an algorithmic improvement: (help sequential algorithm as well)
  - Use interval arithmetic to conservatively test entire set of rays against node bbox (e.g., think of a packet as a beam)
  - Further optimizations possible when all rays share origin
  - Note: value to making packets much bigger than SIMD width!

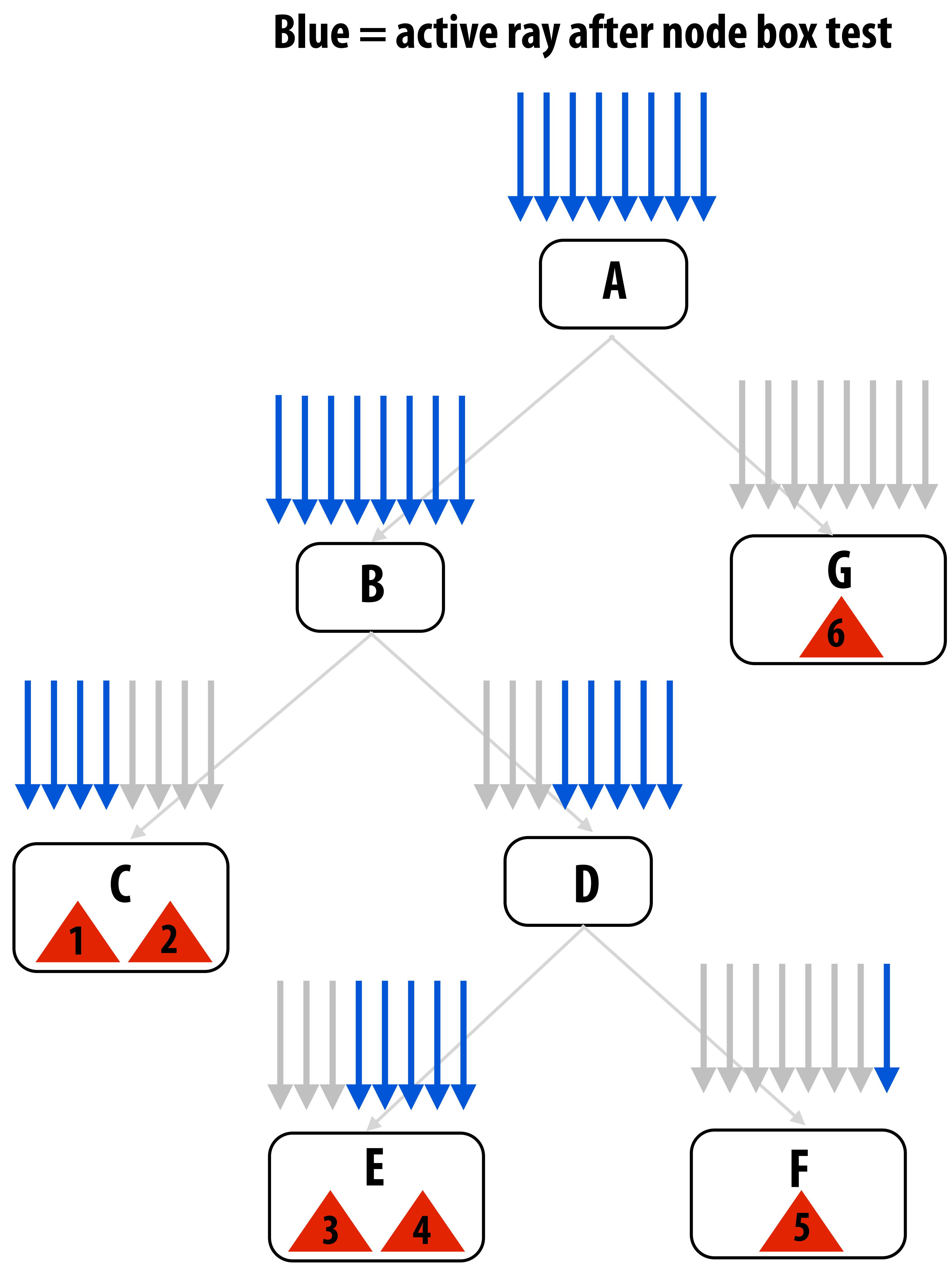
Disadvantages of packets

- If any ray must visit a node, it drags all rays in the packet along with it

  (note contrast with SPMD version: each ray only visits BVH nodes it is required to)

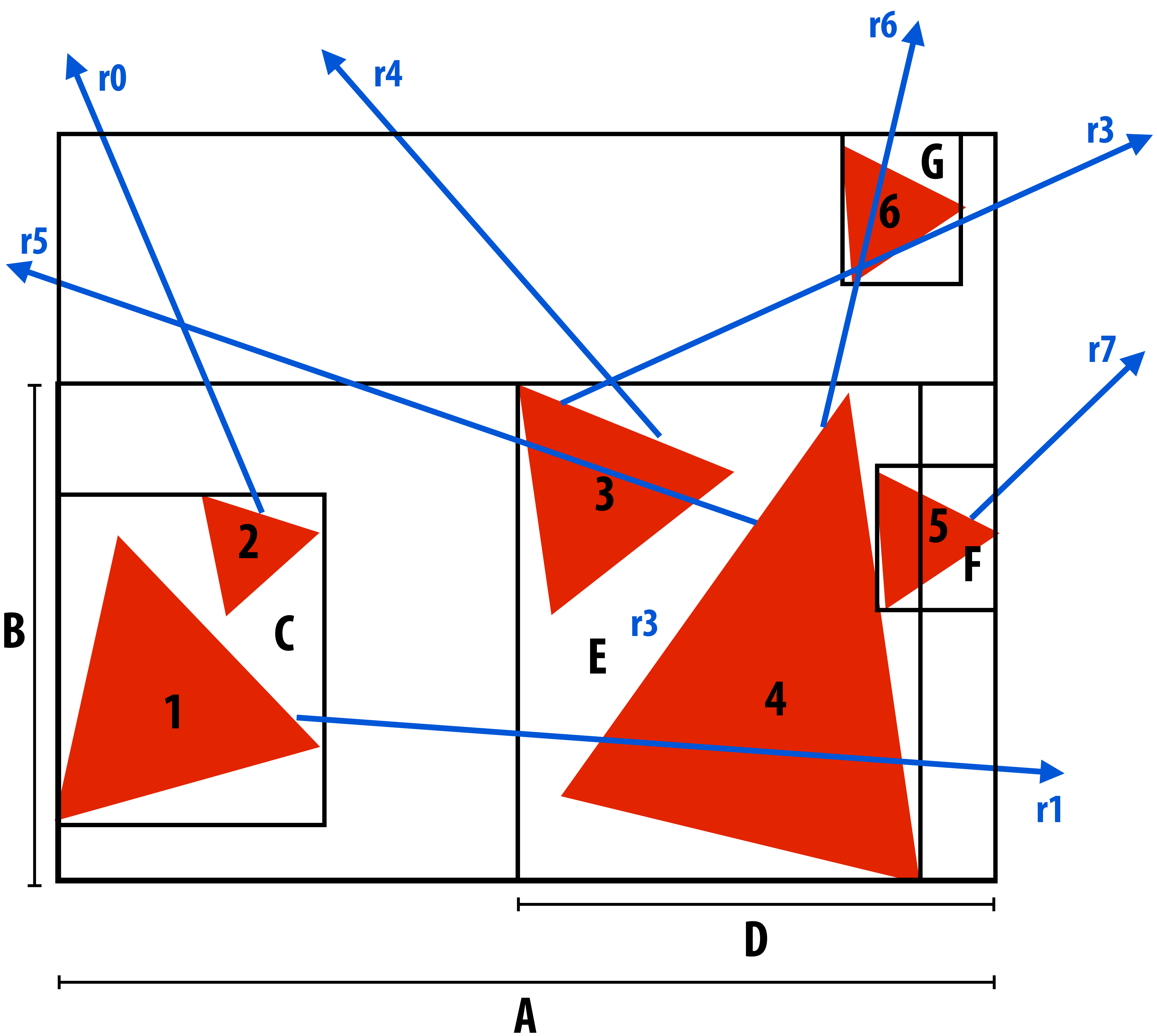
- Loss of efficiency: node traversal, intersection, etc. amortized over less than a packet's worth of rays

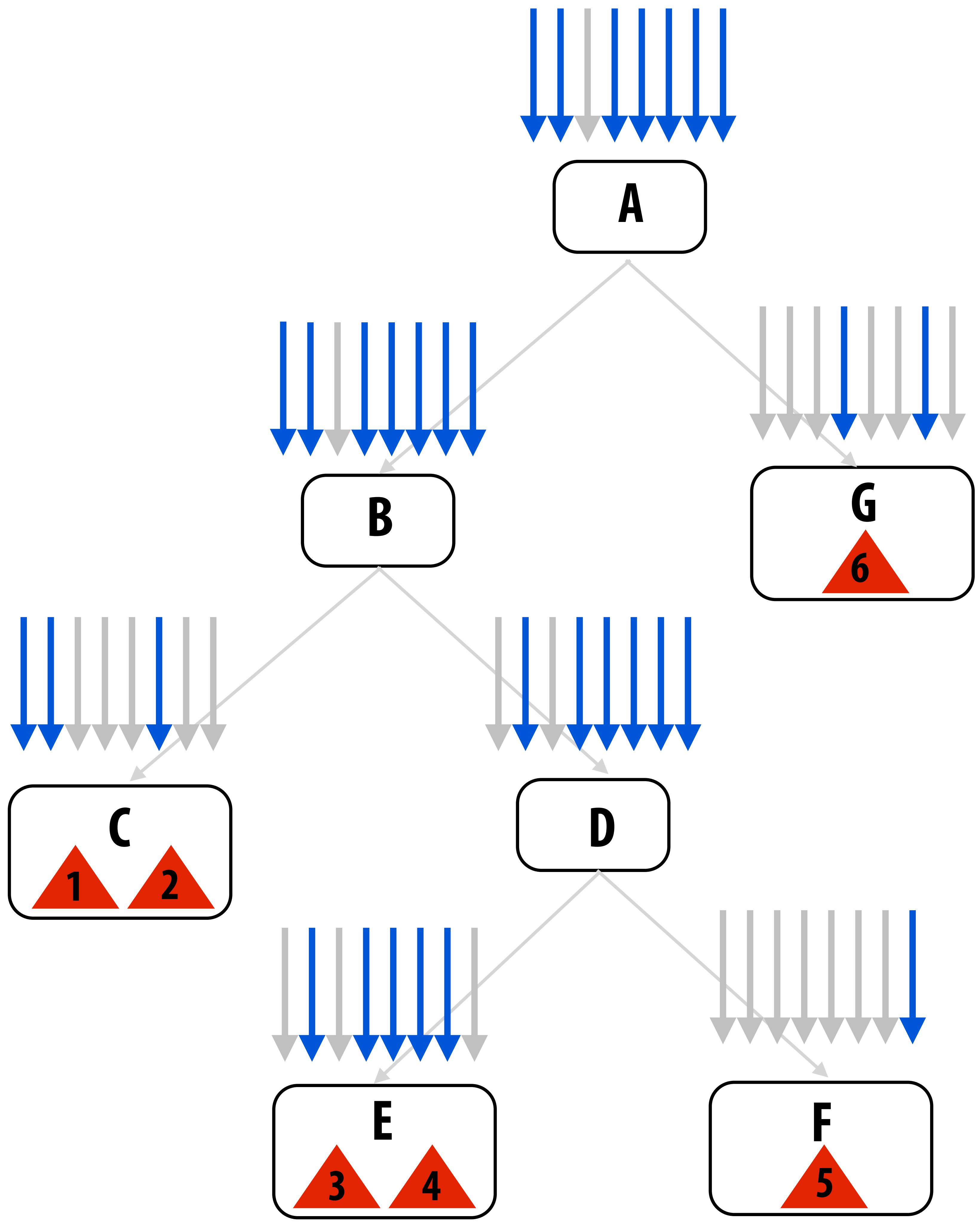
- Not all SIMD lanes doing useful work



Ray packet tracing: incoherent rays

Blue = active ray after node box test





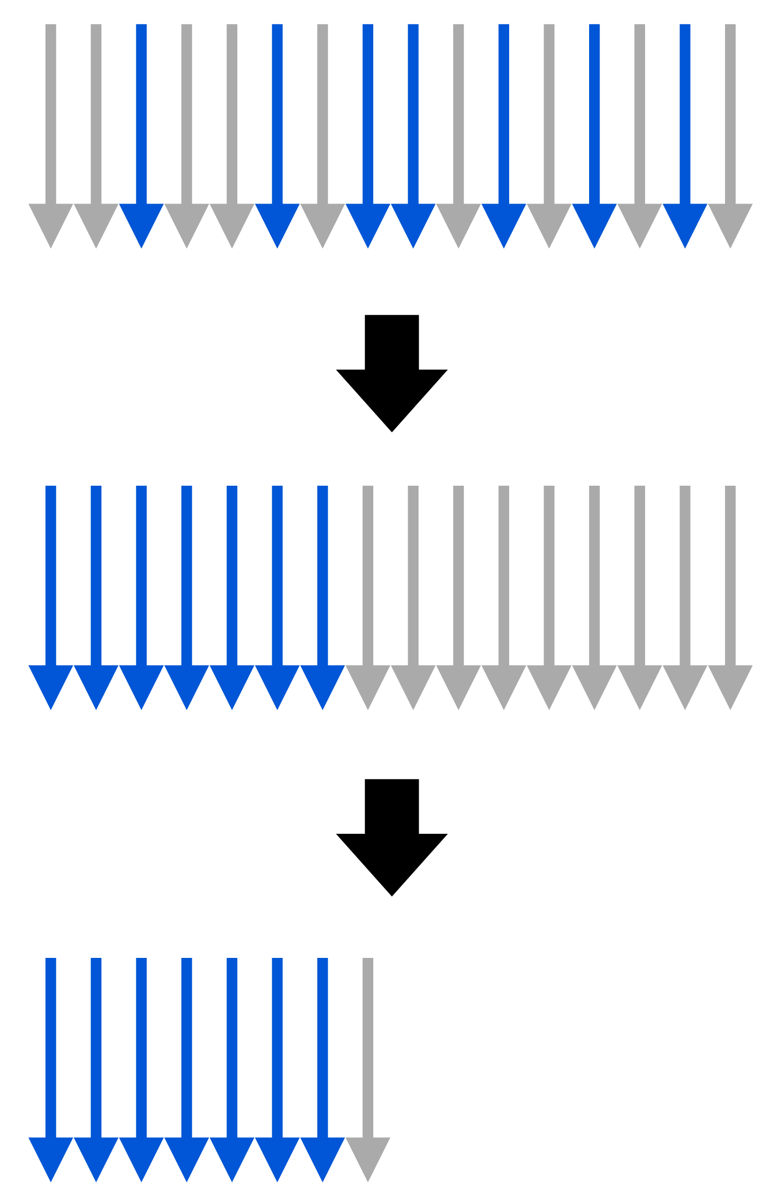
When rays are incoherent, benefit of packets can decrease significantly. This example: packet visits all tree nodes. (All rays visit all tree nodes)

Improving packet tracing with ray reordering

Idea: when packet utilization drops below threshold, resort rays and continue with smaller packet

- Increases SIMD utilization
- Still loses amortization benefits of large packets

Example: 8-wide SIMD processor, 16-ray packets
(2 SIMD instructions required to perform operation on all rays in full packet)



16-ray packet: 7 of 16 rays active

Reorder rays
Recompute intervals/bounds for active rays

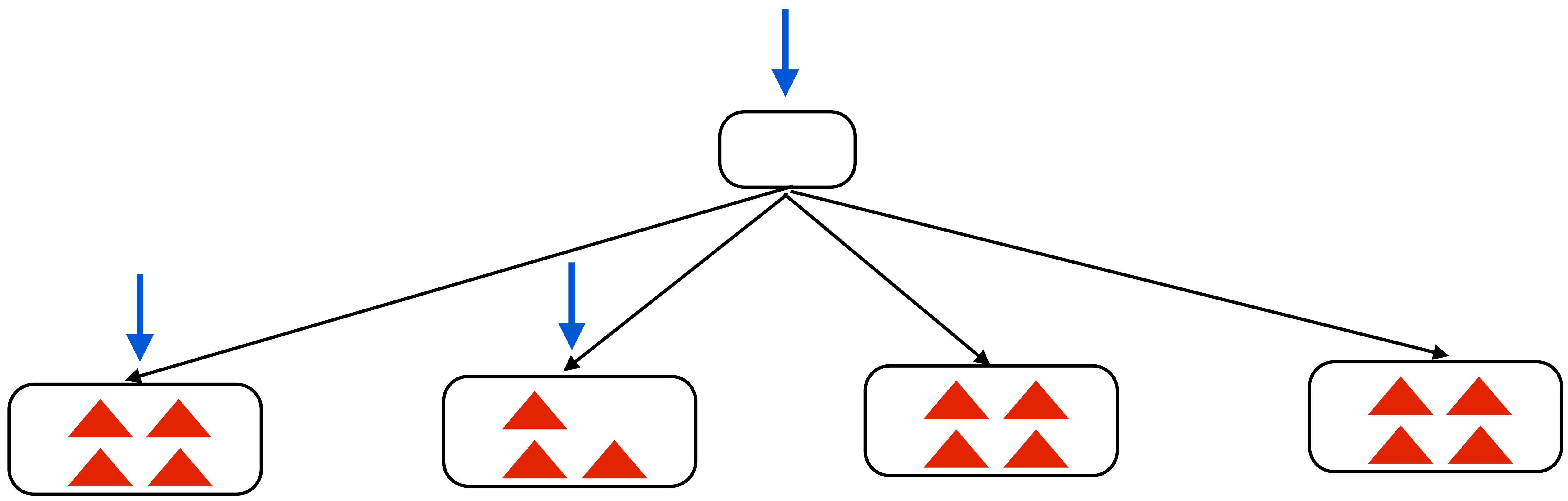
Continue tracing with 8-ray packet:
7 of 8 rays active

(Now: single instruction to perform operation on active rays)

Giving up on packets

- Even with reordering, ray coherence during BVH traversal will diminish
  - Little benefit to packets (can decrease performance compared to single ray code)

- Idea: exploit SIMD execution within <u>single</u> ray-BVH intersection query
  - Interior: use wider-branching BVH
    (test single ray against multiple node bboxes in parallel)
  - Leaf: test ray against multiple triangles in parallel



Packet tracing best practices

- Use large packets for higher levels of BVH
  - Ray coherence always high at the top of the tree (visited by all rays)
- Switch to single ray (intra-ray SIMD) when packet utilization drops below threshold
- Can use dynamic reordering to postpone time of switch
  - Reordering allows packets to provide benefit deeper into tree

Ray tracing data access

- BVH traversal requires a lot of jumping through memory
  - Not predictable by definition (or you have a bad tree)
  - Fine-granularity data access (like Barnes-Hut)
  - Packets amortize cost of node fetches over many rays (fetch data less)

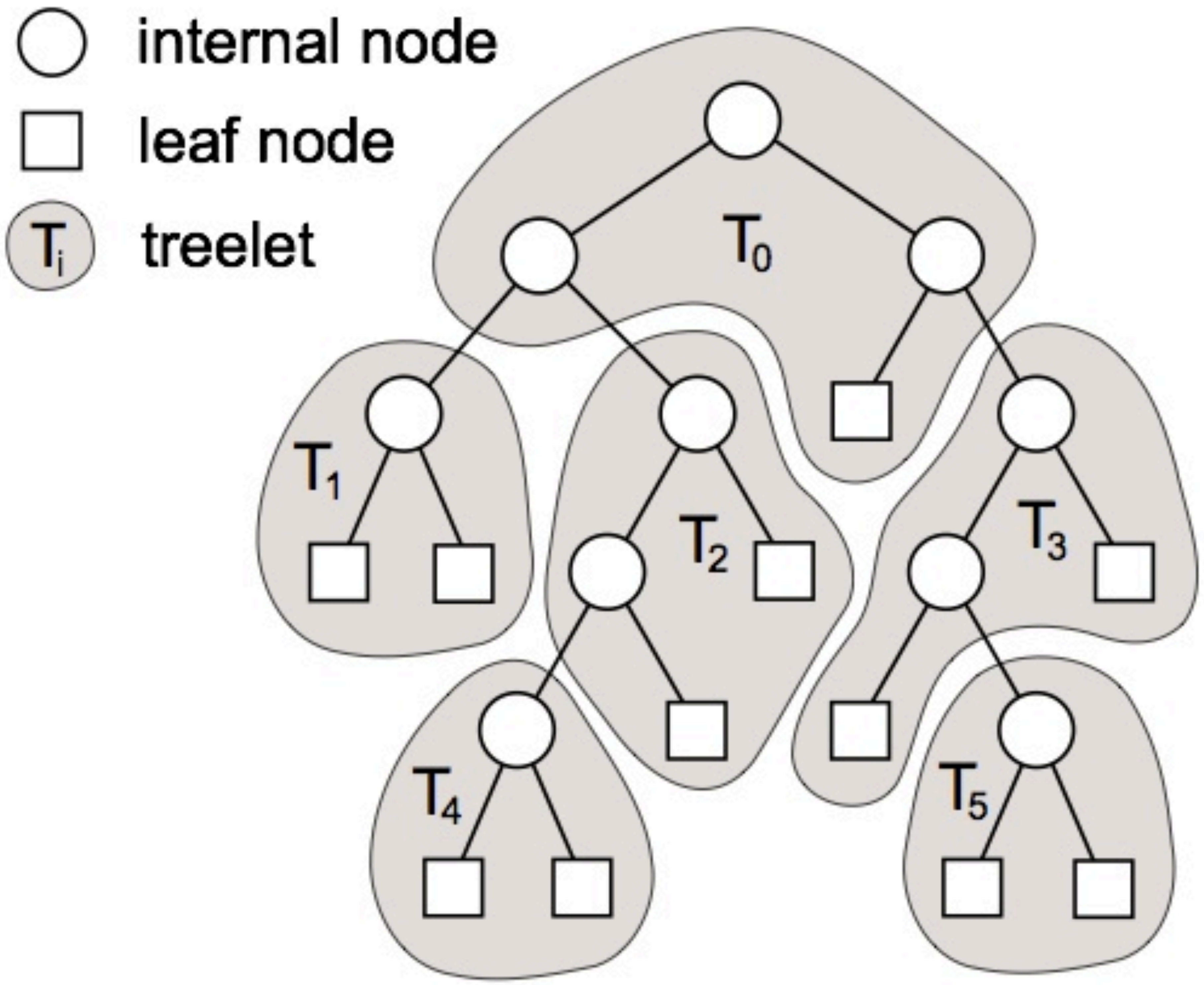
- In form of ray tracing discussed today: tree is read-only during ray traversal
  - Each ray packet processed independently → no synchronization needed between cores
  - Parts of tree replicated in each processor's cache

- Top of tree: high locality (touched by all rays)

- Incoherent ray traversal suffers from poor cache behavior
  - Ray-scene intersection becomes bandwidth bound
  - Large caches typical provide large benefit

Global ray reordering

Idea: batch up rays in the same part of the scene. Process these rays together to increase locality



Partition BVH into treelets
(treelets sized for L1 or L2 cache)

1. **When ray (or packet) enters treelet, add rays to treelet queue**
2. **When treelet queue is sufficiently deep, intersect enqueued rays with treelet**

Costs: global synchronization, extra footprint to store buffered rays

Benefits: increases SIMD coherence, increases locality of tree data access

Summary

- Today: three examples of parallel program optimization
- Key issues when discussing the applications
  - How to balance the work?
  - How to exploit locality inherent in the problem?
  - What synchronization is necessary?