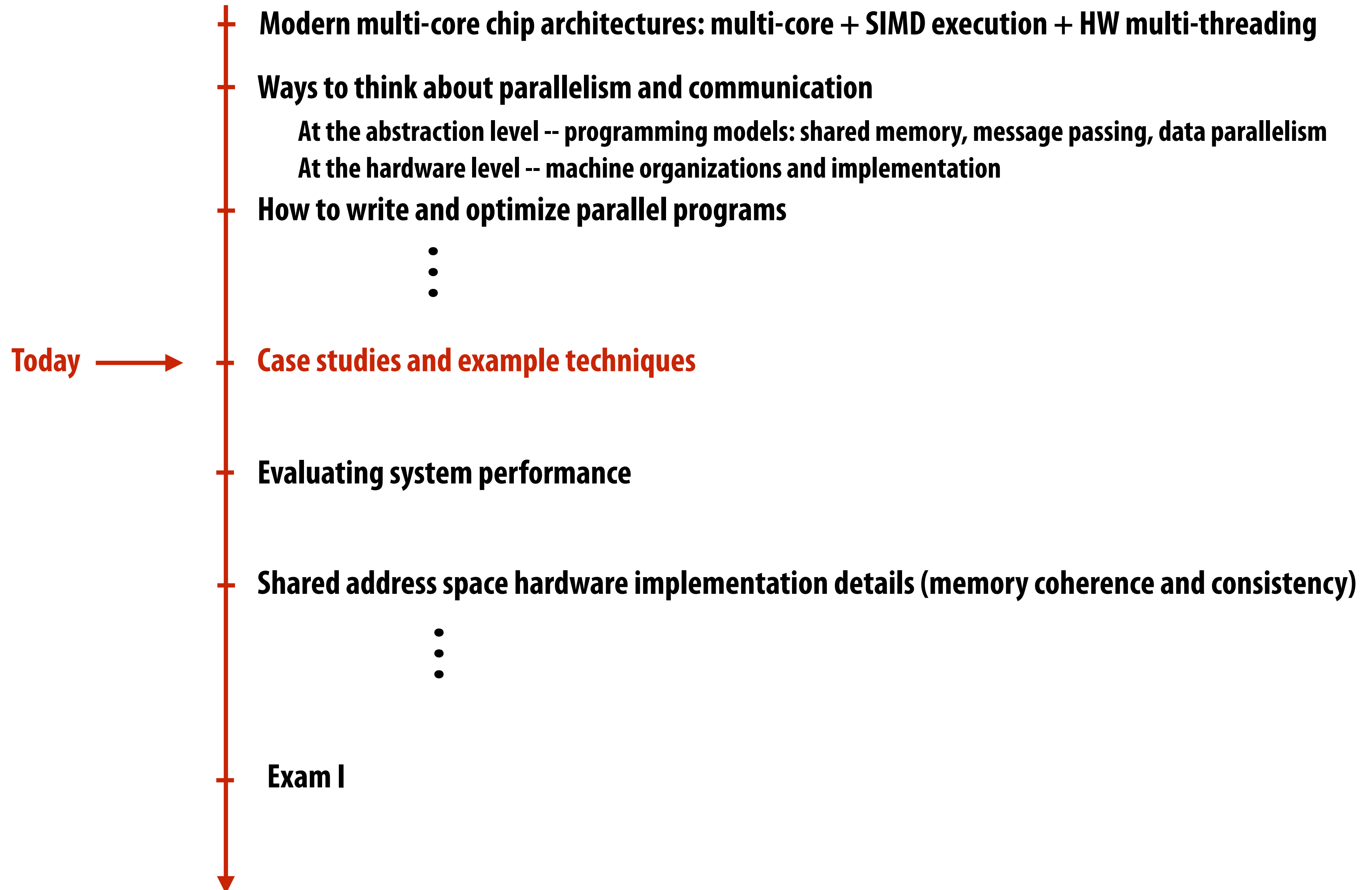
Lecture 8:

Parallel Programming Case Studies

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2016

15-418/618 course road map



Today: case studies!

- Several parallel application examples
  - Ocean simulation
  - Galaxy simulation (Barnes-Hut algorithm)
  - Parallel scan
  - Data-parallel segmented scan (Bonus material!)
  - Ray tracing (Bonus material!)

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

Assumption: shared address space

- **For the purposes of today's lecture I encourage you to think about the example applications in the context of a large NUMA shared address space machine.**

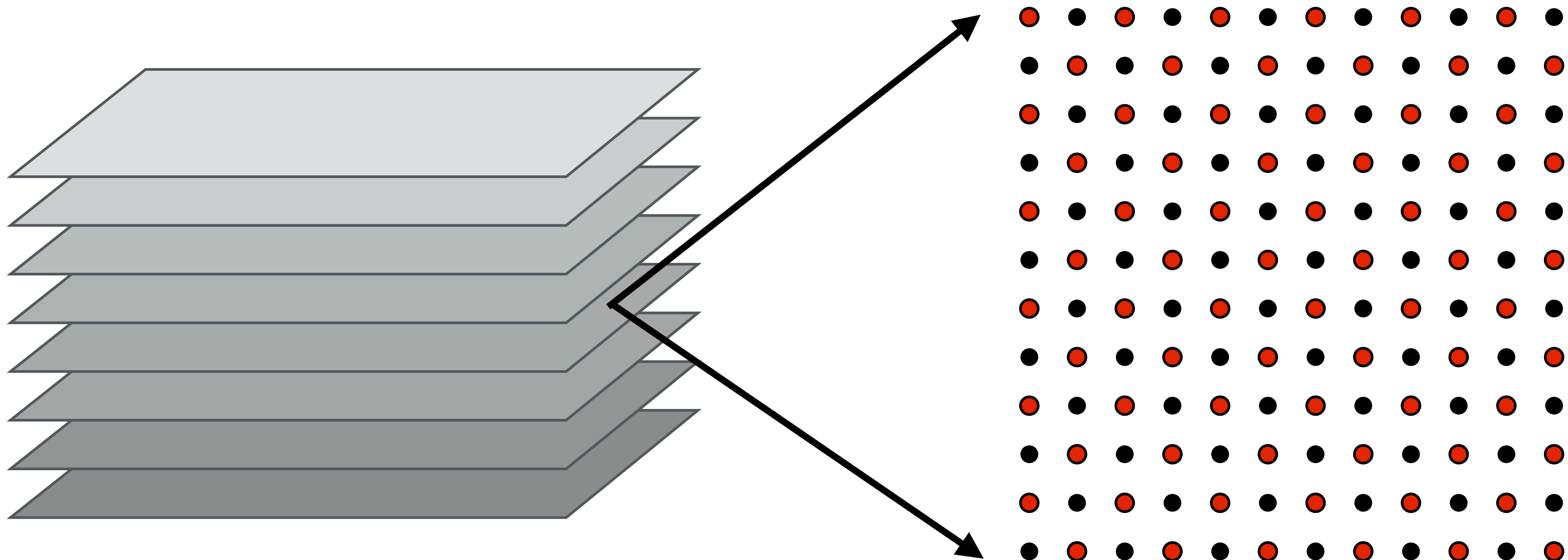
  (single address space, but each processor can access a local region of the address space more quickly)

- **But issues we discuss certainly also arise in a distributed address space setting.**

Simulation of Ocean Currents (grid-based solver)

Example taken from: Culler, Singh, and Gupta

Simulating of ocean currents

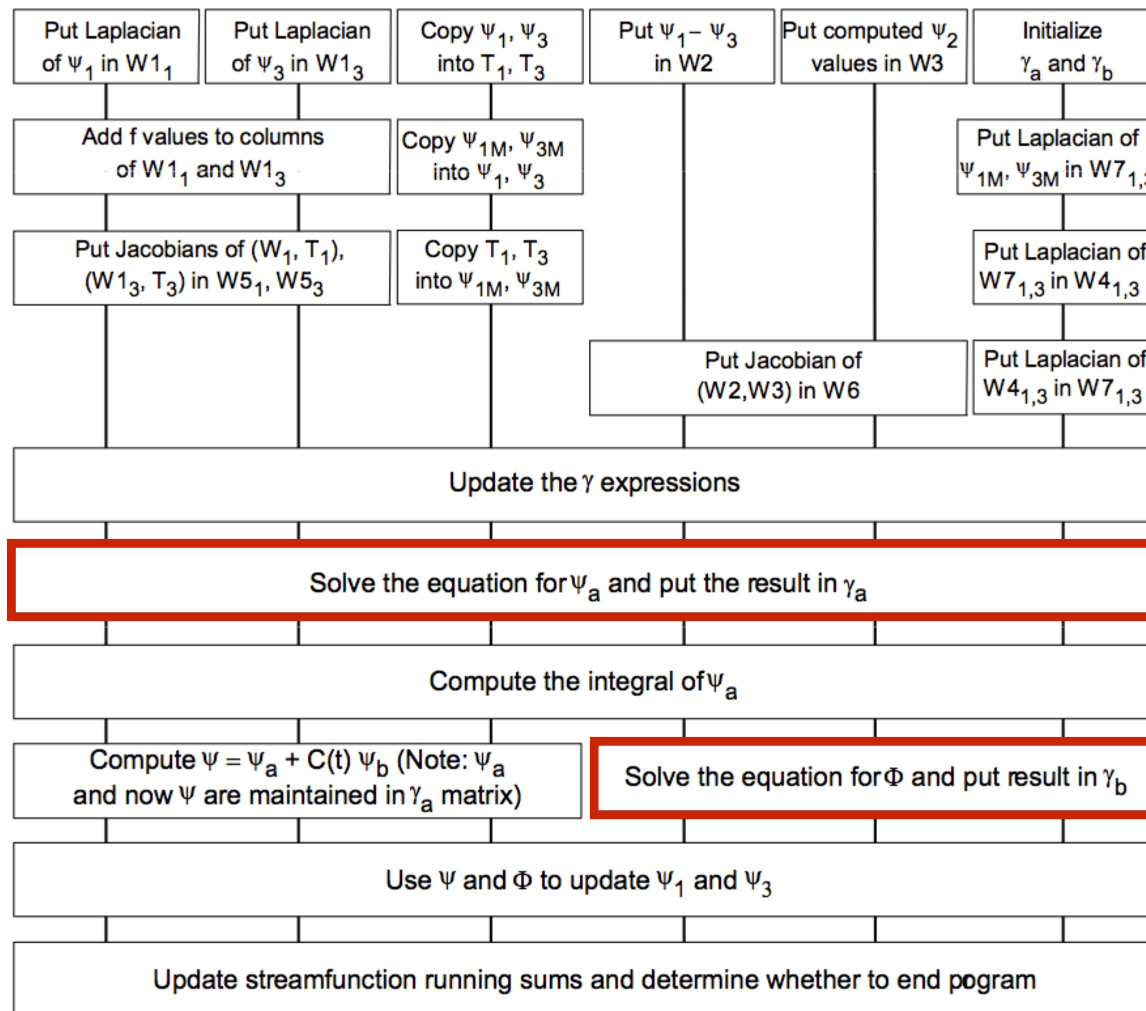


- Discretize 3D ocean volume into slices represented as 2D grids
- Discretize time evolution of ocean: Δt
- High accuracy simulation requires small Δt and high resolution grids

Figure credit: Culler, Singh, and Gupta

Where are the dependencies?

Dependencies in one time step of ocean simulation



Boxes correspond to computations on grids

Lines express dependencies between computations on grids

The "grid solver" example corresponds to these parts of the application

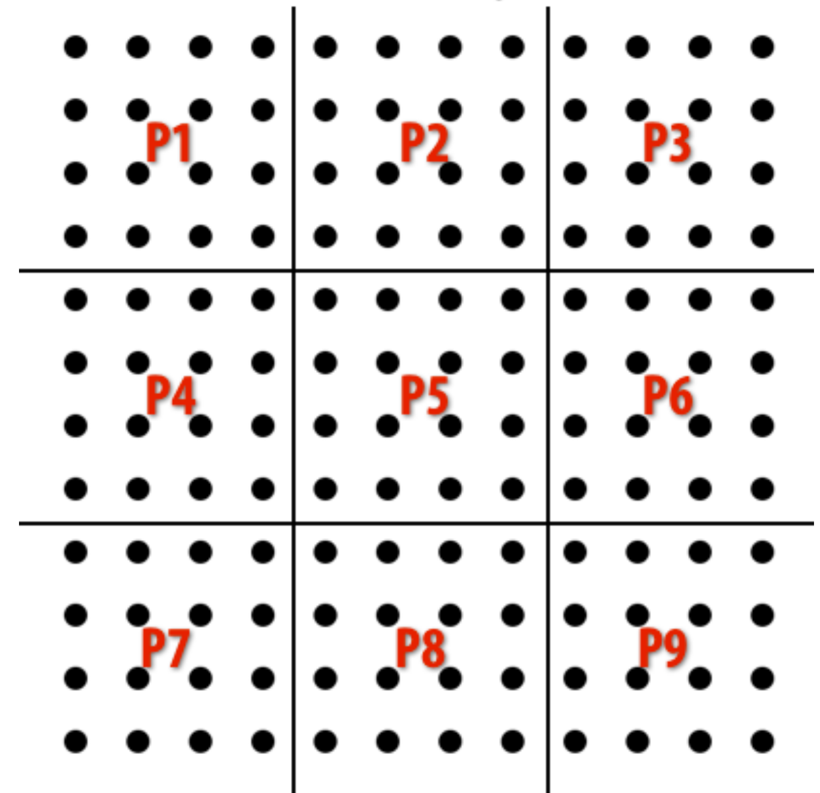
Parallelism within a grid (data-parallelism) and across operations on the different grids.
The implementation only leverages data-parallelism (for simplicity)

Figure credit: Culler, Singh, and Gupta

Ocean implementation details

Recall shared-memory implementation discussed in previous classes:

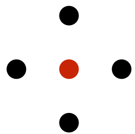
- **Decomposition:**
  - Spatial partitioning of grid: each processor receives 2D tile of grid
- **Assignment**
  - Static assignment of tiles to processors
- **Synchronization**
  - Barriers (separate each pass over grid is a different phase of computation)
  - Locks for mutual exclusion when updating shared variables (atomic update of 'diff')



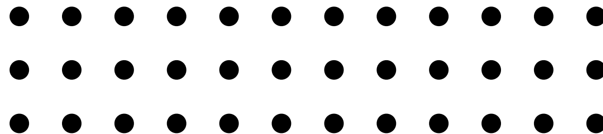
Another question to ask: what are the critical working sets?

1. Local neighborhood for cell
2. Three rows of a processor's local partition of grid
3. Processor's local partition of grid

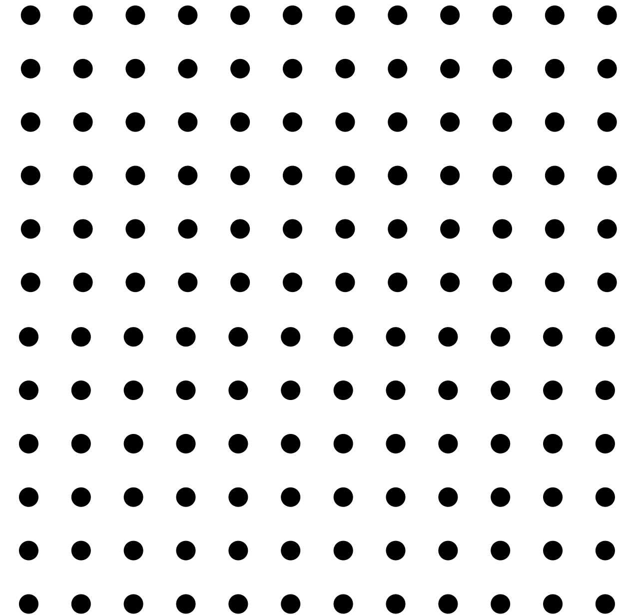
1.



2.



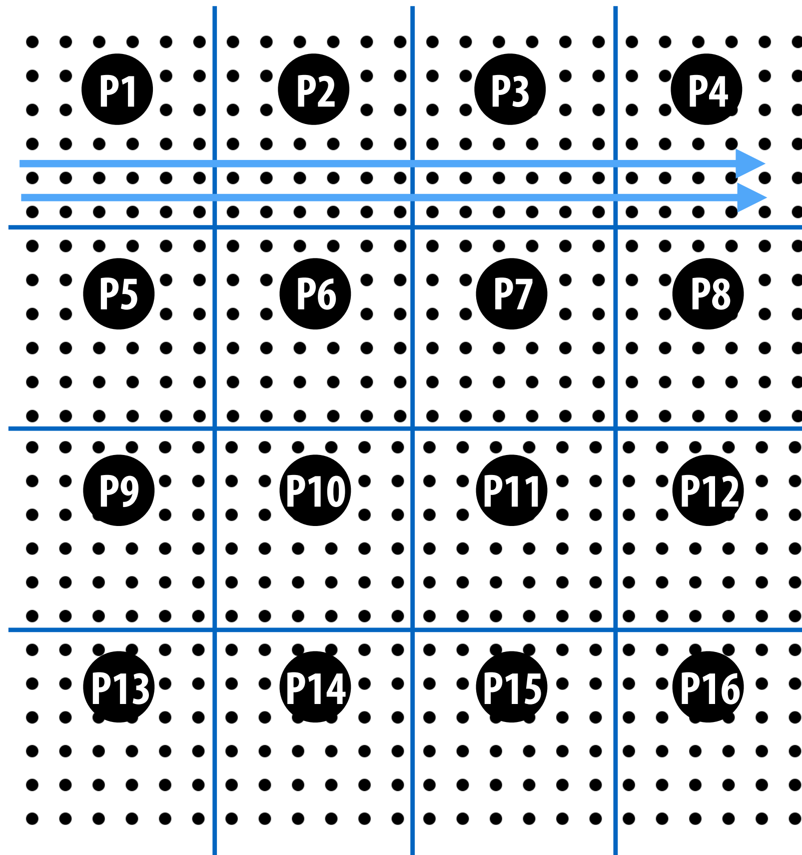
3.



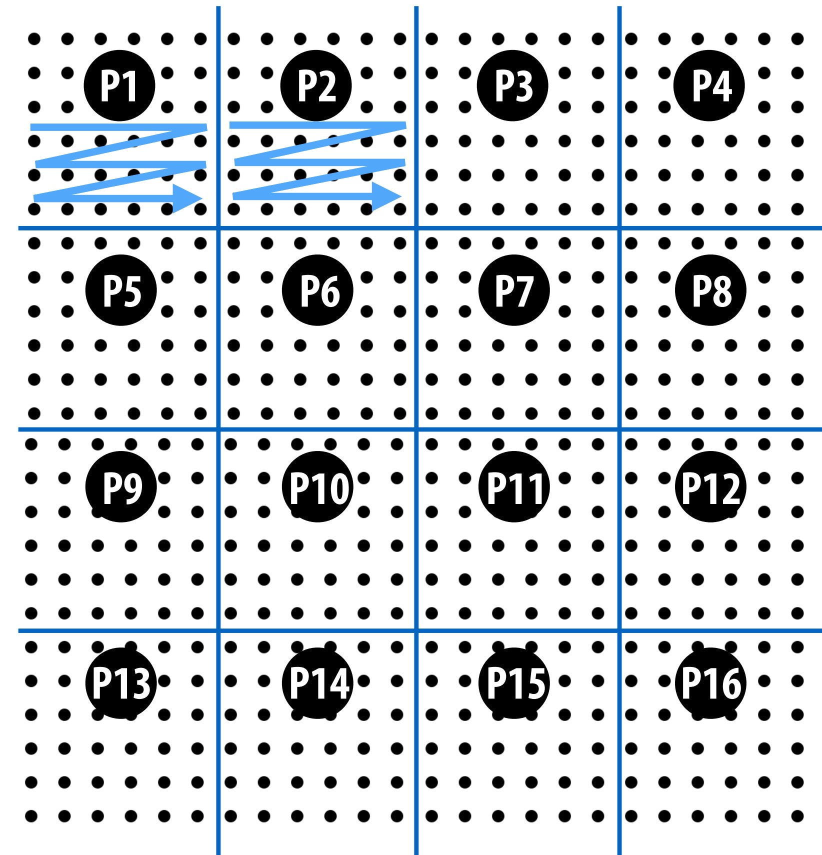
Recall: two layouts of 2D grid in address space

(Blue lines indicate consecutive memory addresses)

2D, row-major array layout

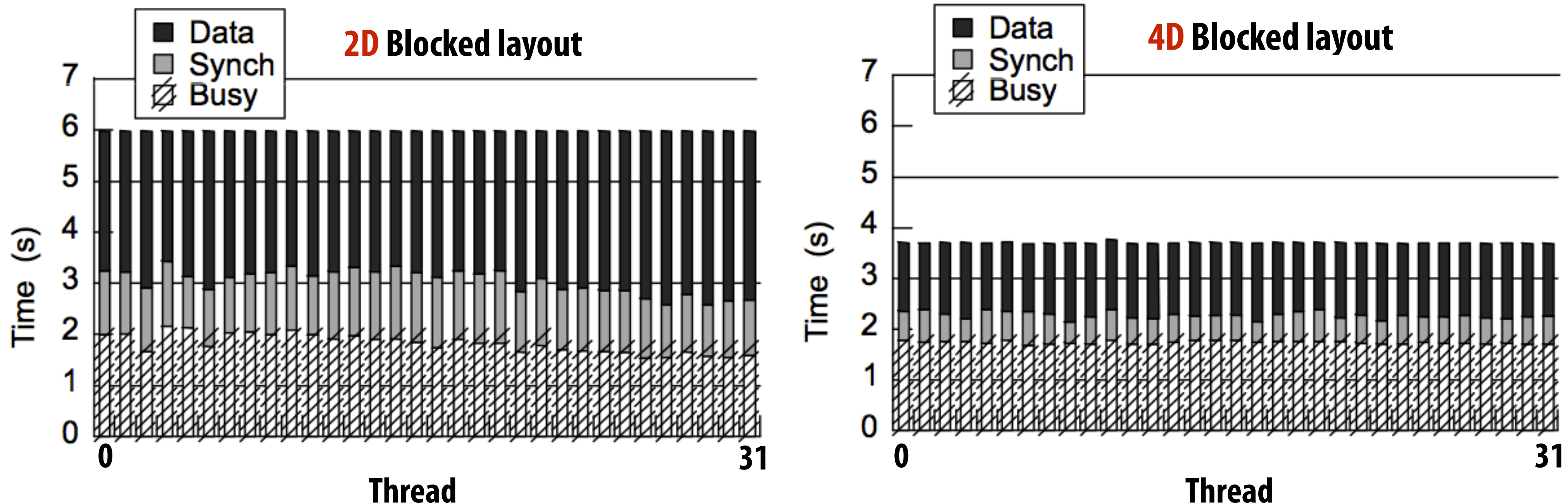


4D array layout (block-major)



Ocean: execution time breakdown

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



Observations:

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

Figure credit: Culler, Singh, and Gupta

Galaxy Evolution using Barnes Hut

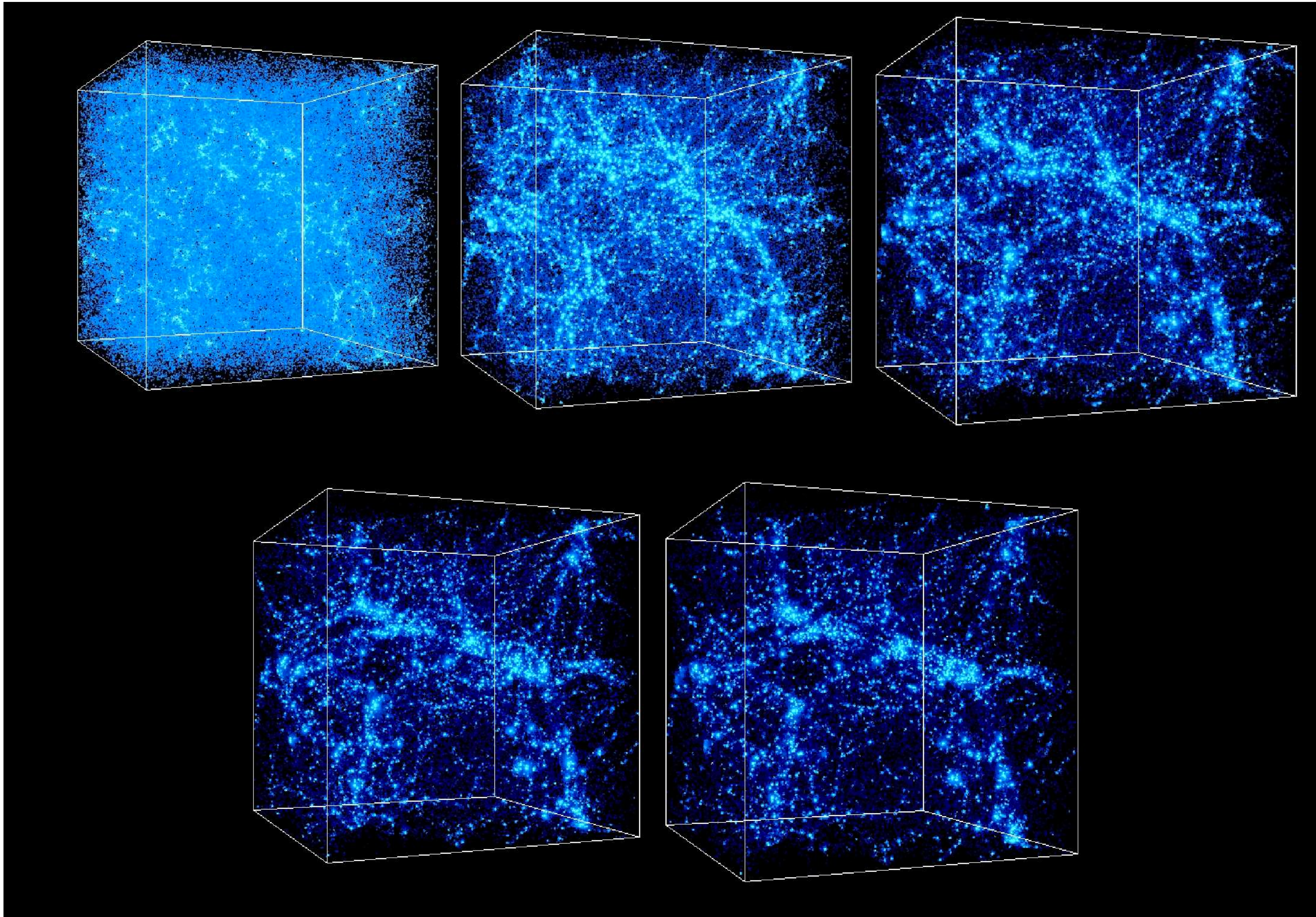
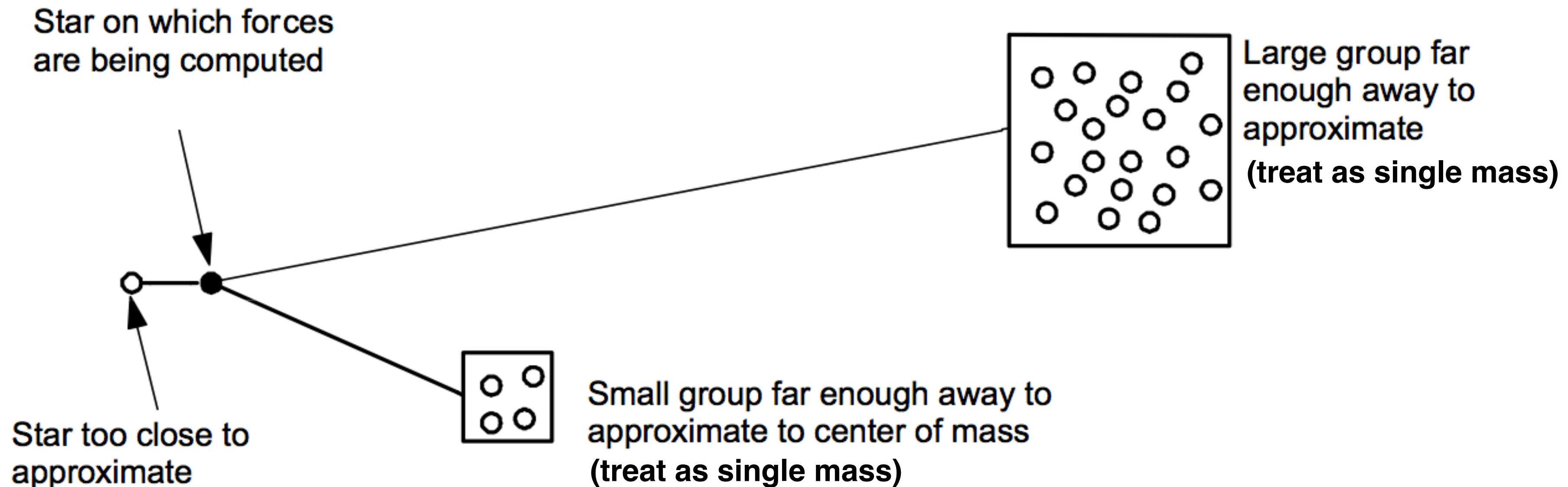


Image credit: http://www.lsw.uni-heidelberg.de/users/mcamenzi/images/Universe_Box.gif
Example taken from: Culler, Singh, and Gupta, Chapter 3

Galaxy evolution

Barnes-Hut algorithm



- Represent galaxy as a collection of N particles (think: particle = star)
- Compute forces on each particle due to gravity
  - Naive algorithm is $O(N^2)$ — all particles interact with all others (gravity has infinite extent)
  - Magnitude of gravitational force falls off with distance (so algorithms approximate forces from far away stars to gain performance)
  - Result is an O(NlgN) algorithm for computing gravitational forces between all stars

Barnes-Hut tree

Spatial Domain

Quad-Tree Representation of Bodies

- Leaf nodes are star particles
- Interior nodes store center of mass + aggregate mass of all child bodies
- To compute forces on each body, traverse tree... accumulating forces from all other bodies
  - Compute forces using aggregate interior node if $L/D < \Theta$, else descend to children
- Expected number of nodes touched ~ $\lg N / \Theta^2$

Barnes-Hut application structure

```
for each time step in simulation:
   build tree structure
   compute (aggregate mass, center-of-mass) for interior nodes
   for each particle:
      traverse tree to accumulate gravitational forces
      update particle position based on gravitational forces
```

Challenges:

- Amount of work per body is non-uniform, communication pattern is non-uniform (depends on the local density of bodies)
- 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 that are near in space require similar data to compute forces — it seems smart to co-locate these computations!)

Work 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 (the application profiles itself)
    - Cheap to compute. Just increment local per-body counters
    - Use values to periodically recompute assignment

Assignment using cost zones

- Leverage locality inherent in tree
- Compute total work estimate W for all bodies (computed by summing per-body costs)
- Each processor is assigned W/P of the total work (P = num processors)
- Each processor performs depth-first (post-order) traversal of tree (accumulates work seen so far)
- Processor P_i responsible for processing bodies corresponding to work: iW/P to $(i+1)W/P$
- Each processor can independently compute its assignment of bodies. (The only synchronization required is the sum reduction to compute total amount of work = W)

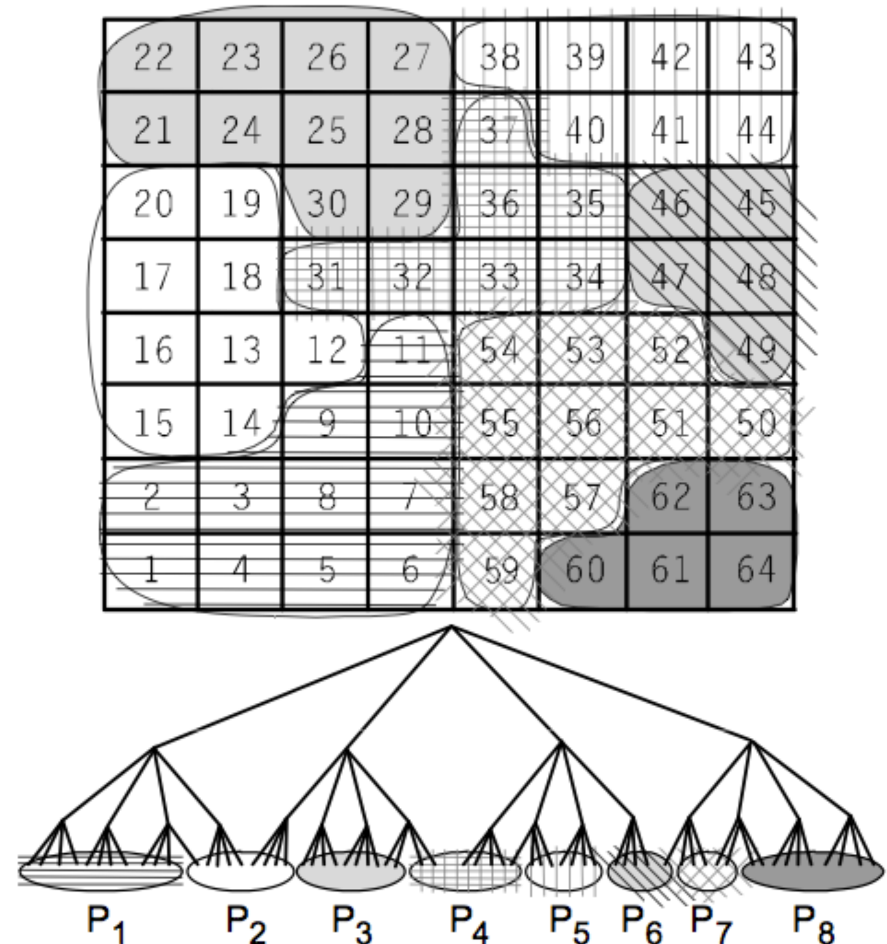
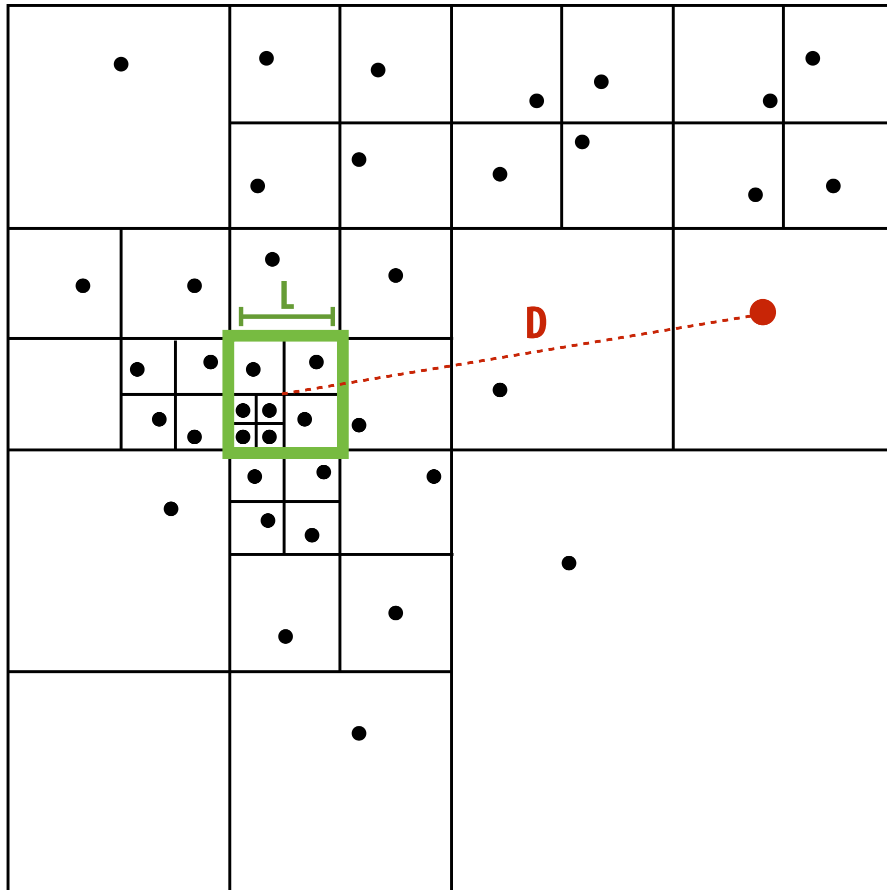
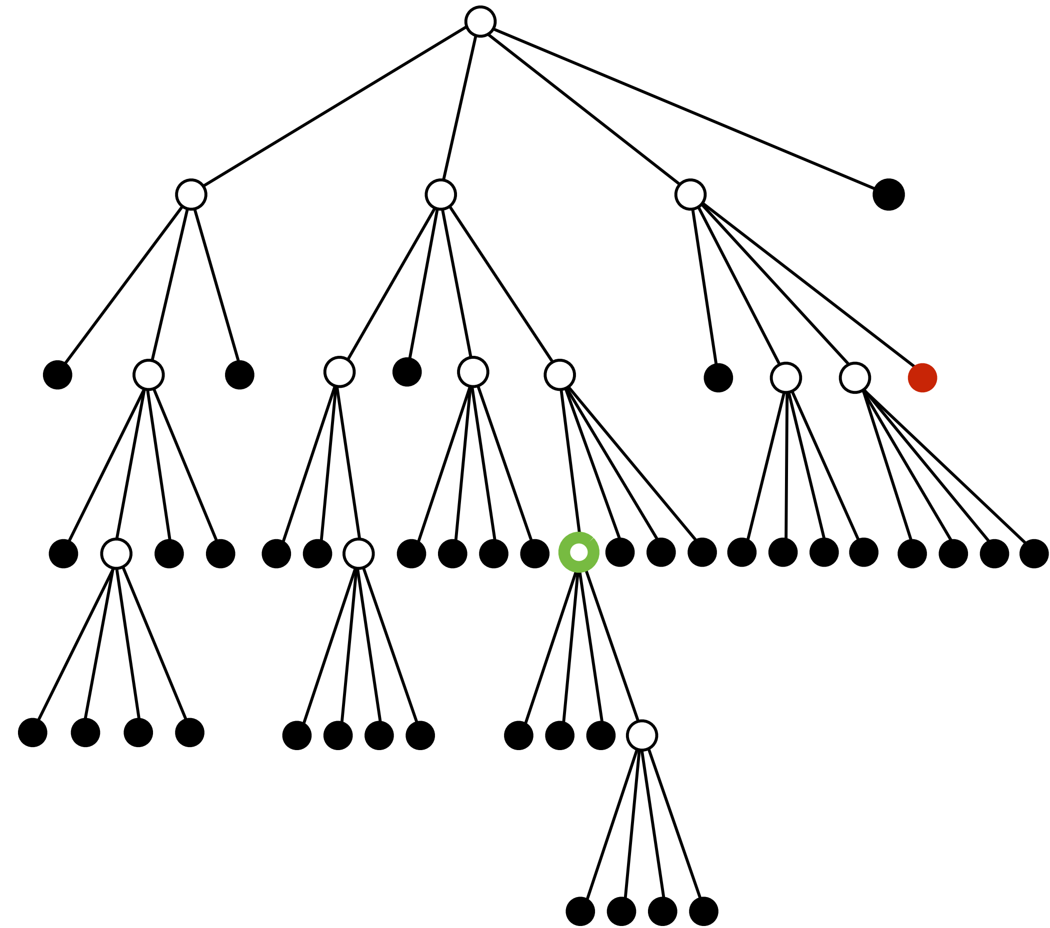


Figure credit: Culler, Singh, and Gupta

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: ~ lg N / Θ^2
  - Computation has high locality: consecutively processed bodies are nearby, so processing touches almost exactly the same nodes!

Barnes-hut: data distribution

- Cost zones technique computes a good work assignment. What about data distribution?
- Difficult to distribute data
  - Work assignment changes with time: would have to dynamically redistribute all simulation data
  - Data accessed at fine granularity (single tree node)
- Luckily: high temporal locality
  - Bodies assigned to same processor are nearby in space, so tree nodes accessed during force computations are very similar.
  - Data for traversal already 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
  - Implementation uses static distribution (interleave particles throughout the machine)

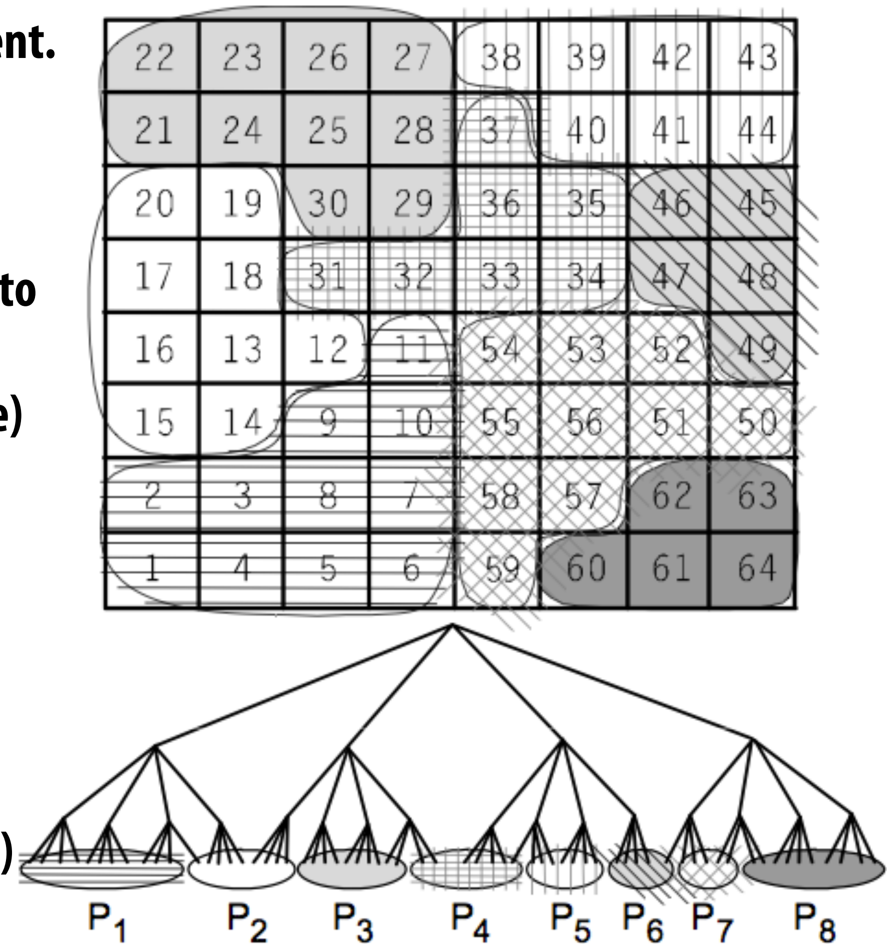
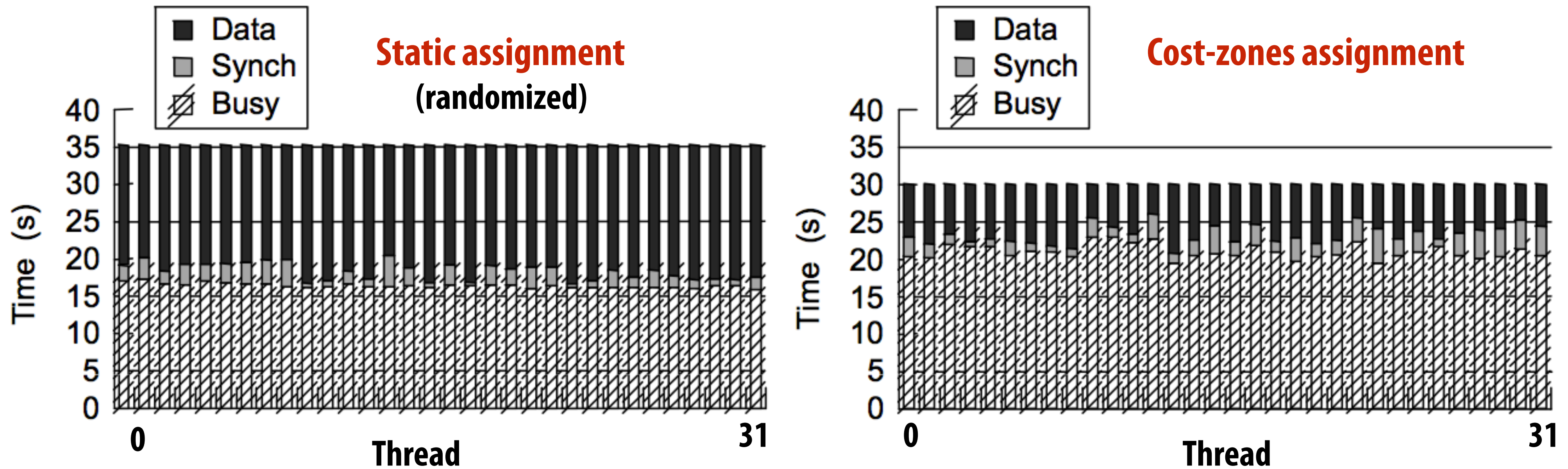


Figure credit: Culler, Singh, and Gupta

Barnes-hut: execution time

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



- **Load balance is good even with static assignment because of random assignment**
  - **On average, each processor does approximately the same amount of work**
- **But random assignment yields poor locality**
  - **Significant amount of inherent communication**
  - **Significant amount of artifactual communication (fine-grained accesses to tree nodes)**
- **Common tension: work balance vs. locality (cost-zones get us both!)**
  (similar to work balance vs. synchronization trade-offs in "work distribution" lecture)

Figure credit: Culler, Singh, and Gupta

Summary

- Today so far: two 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?

Parallel Scan

Data-parallel scan

let $A = [a_0, a_1, a_2, a_3, ..., a_{n-1}]$

let $\oplus$ **be an associative binary operator with identity element** I

`scan_inclusive(⊕, A)` $= [a_0,\ a_0 \oplus a_1,\ a_0 \oplus a_1 \oplus a_2,\ ...$

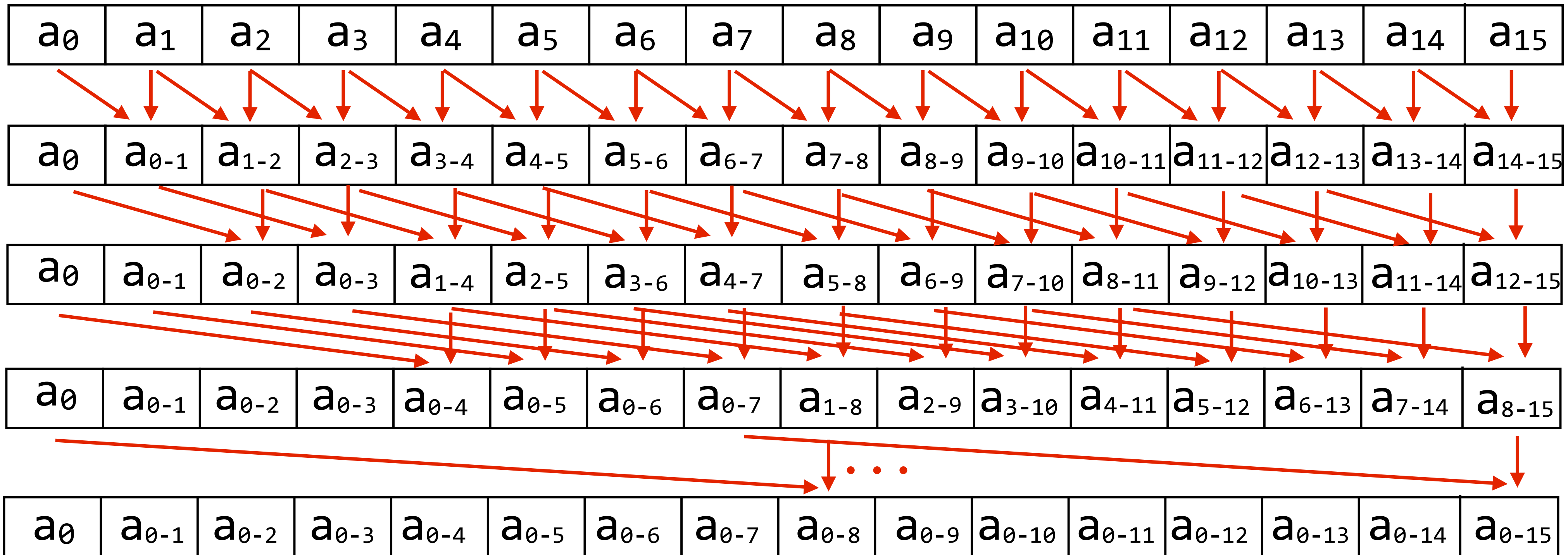
`scan_exclusive(⊕, A)` $= [I,\ a_0,\ a_0 \oplus a_1,\ ...$

If operator is +, then `scan_inclusive(+,A)` **is a prefix sum**

`prefix_sum(A)` $= [a_0,\ a_0 + a_1,\ a_0 + a_1 + a_2,\ ...$

Data-parallel inclusive scan

(Subtract original vector to get exclusive scan result: not shown)



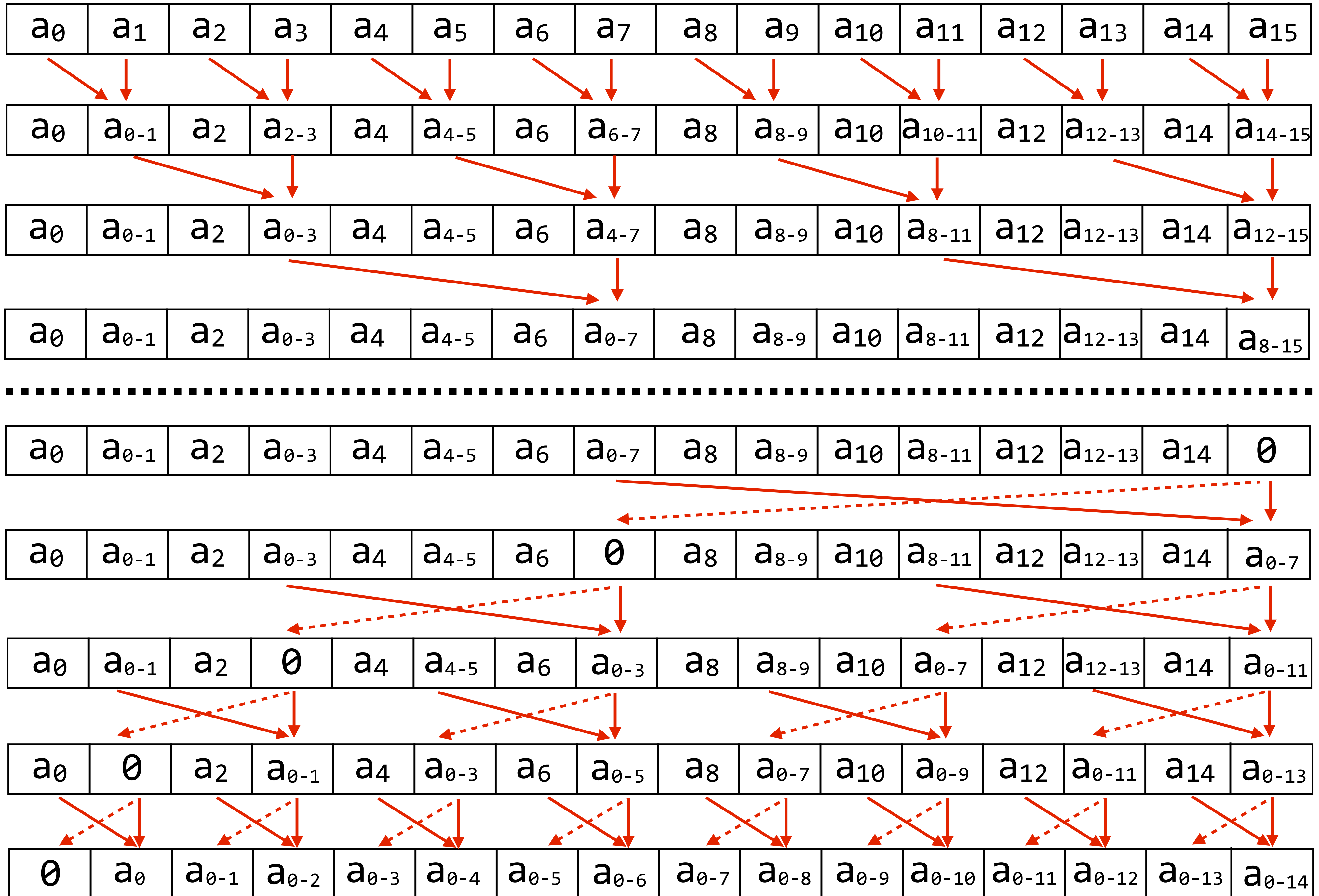
* not showing all dependencies in last step

Work: O(N lg N)

Inefficient compared to sequential algorithm!

Span: O(lg N)

Work-efficient parallel exclusive scan (O(N) work)



Work efficient exclusive scan algorithm

(with $\oplus$ = "+")

Up-sweep:

```
for d=0 to (log₂n - 1) do
   forall k=0 to n-1 by 2^(d+1) do
     a[k + 2^(d+1) - 1] = a[k + 2^d - 1] + a[k + 2^(d+1) - 1]
```

Down-sweep:

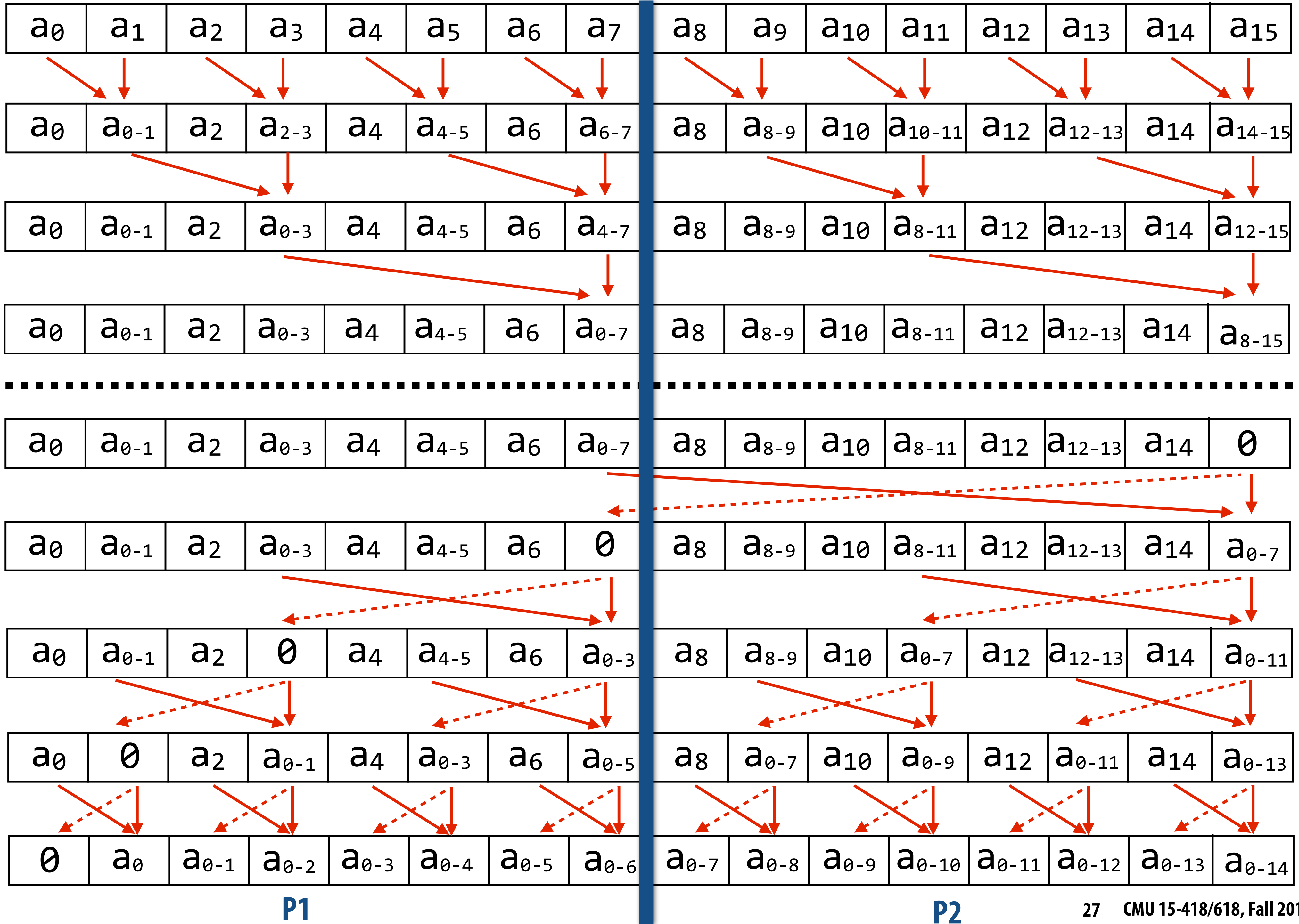
```
x[n-1] = 0
for d=(log₂n - 1) down to 0 do
   forall k=0 to n-1 by 2^(d+1) do
     tmp = a[k + 2^d - 1]
     a[k + 2^d - 1] = a[k + 2^(d+1) - 1]
     a[k + 2^(d+1) - 1] = tmp + a[k + 2^(d+1) - 1]
```

Work: O(N) **(but what is the constant?)**

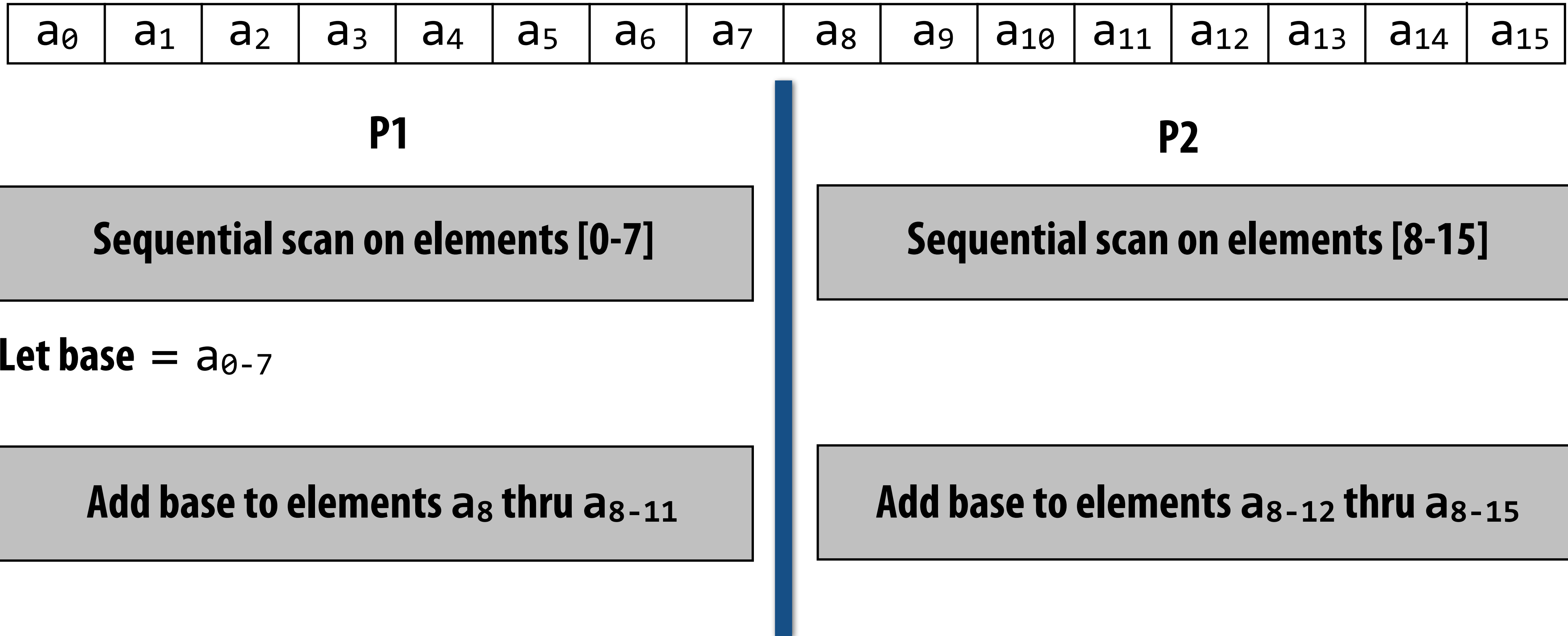
Span: O(lg N) **(but what is the constant?)**

Locality: ??

Now consider scan implementation on just two cores



Exclusive scan: two processor implementation



Work: O(N) (but constant is now only 1.5)

Data-access:

- Very high spatial locality (contiguous memory access)
- P1's access to a_8 through a_{8-11} may be more costly on large NUMA system, but on small-scale system access likely same cost as from P2

Exclusive scan: SIMD implementation (in CUDA)

Example: perform exclusive scan on 32-element array: SPMD program, assume 32-wide SIMD execution

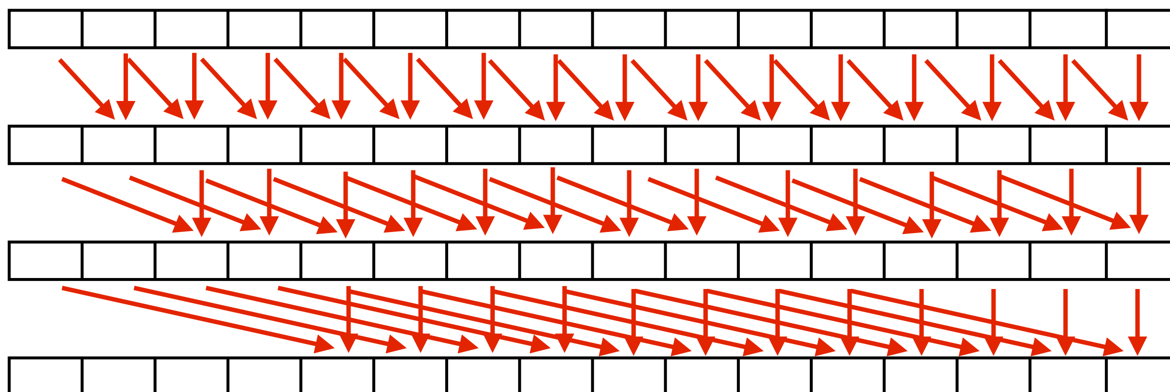
When `scan_warp` is run by a group of 32 CUDA threads, each thread returns the exclusive scan result for element `idx`

(also: upon completion `ptr[]` stores inclusive scan result)

CUDA thread index of caller

```
__device__ int scan_warp(volatile int *ptr, const unsigned int idx)
{
   const unsigned int lane = idx & 31; // index of thread in warp (0..31)

   if (lane >= 1)  ptr[idx] = ptr[idx - 1]  +  ptr[idx];
   if (lane >= 2)  ptr[idx] = ptr[idx - 2]  +  ptr[idx];
   if (lane >= 4)  ptr[idx] = ptr[idx - 4]  +  ptr[idx];
   if (lane >= 8)  ptr[idx] = ptr[idx - 8]  +  ptr[idx];
   if (lane >= 16) ptr[idx] = ptr[idx - 16] +  ptr[idx);

   return (lane > 0) ? ptr[idx-1] : 0;
}
```

Work: ??



Exclusive scan: SIMD implementation (in CUDA)

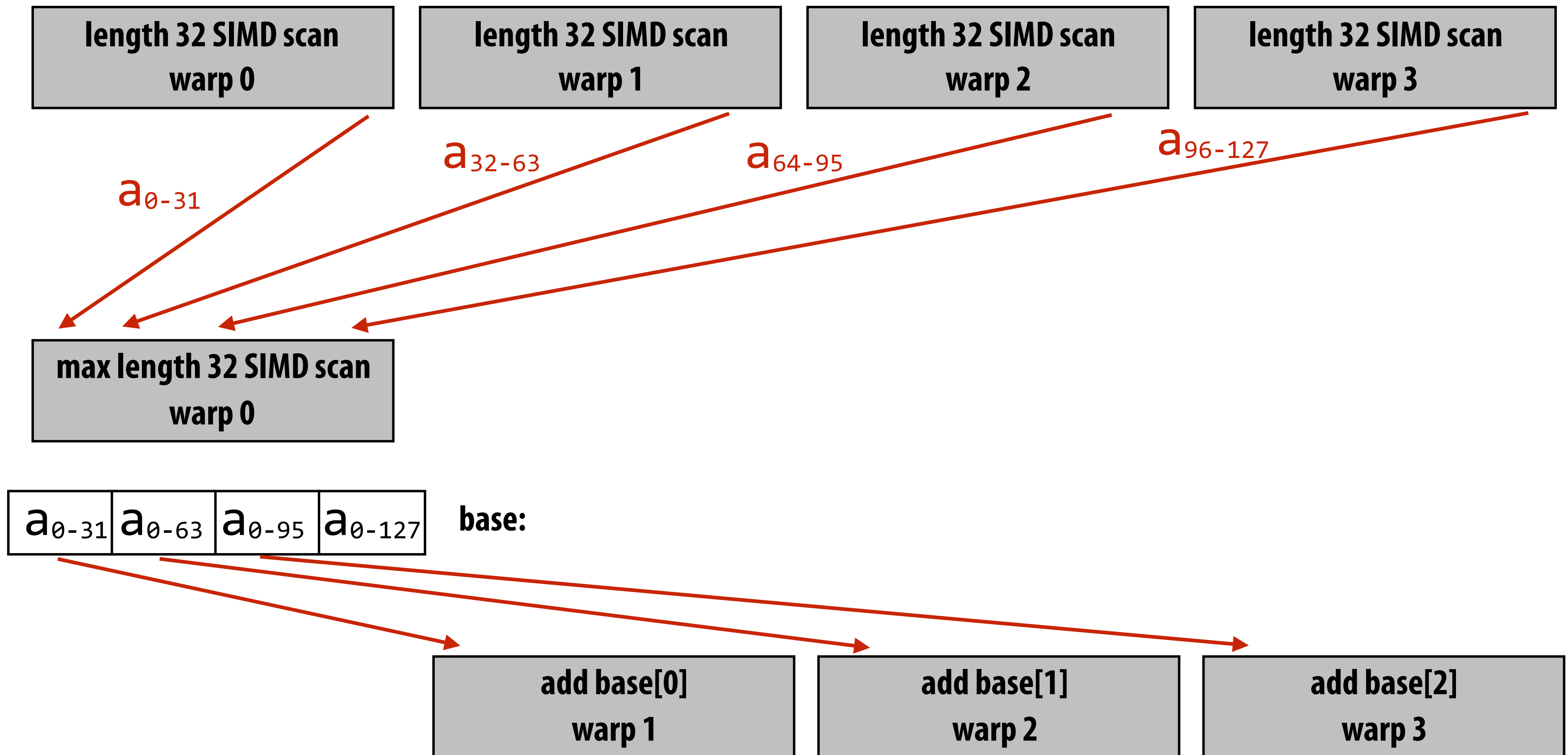
```
__device__ int scan_warp(volatile int *ptr, const unsigned int idx)
{
   const unsigned int lane = idx & 31; // index of thread in warp (0..31)

   if (lane >= 1)  ptr[idx] = ptr[idx - 1]  +  ptr[idx];
   if (lane >= 2)  ptr[idx] = ptr[idx - 2]  +  ptr[idx];
   if (lane >= 4)  ptr[idx] = ptr[idx - 4]  +  ptr[idx];
   if (lane >= 8)  ptr[idx] = ptr[idx - 8]  +  ptr[idx];
   if (lane >= 16) ptr[idx] = ptr[idx - 16] +  ptr[idx];

   return (lane > 0) ? ptr[idx-1] : 0;
}
```

Work: N lg(N)

Work-efficient formulation of scan is not beneficial in this context because it results in low SIMD utilization. It would require more than 2x the number of instructions as the implementation above!

Building scan on larger array

Example: 128-element scan using four-warp thread block



Multi-threaded, SIMD implementation

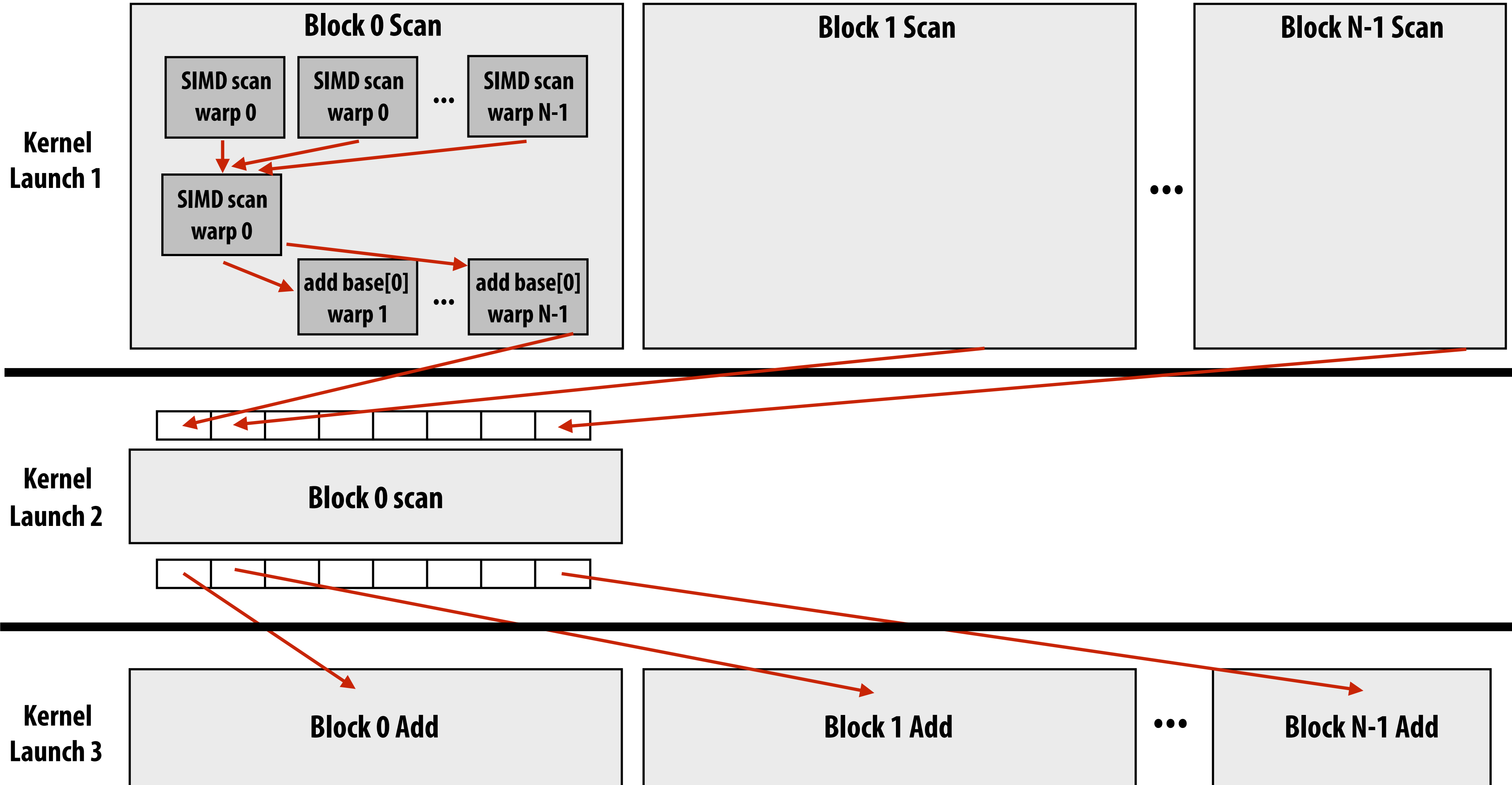
Example: cooperating threads in a CUDA thread block perform scan

We provided similar code in assignment 2.
Code assumes length of array given by `ptr` is same as number of threads per block.

```
__device__ void scan_block(volatile int *ptr, const unsigned int idx)
{
   const unsigned int lane = idx & 31;     // index of thread in warp (0..31)
   const unsigned int warp_id = idx >> 5;  // warp index in block

   int val = scan_warp(ptr, idx);                   // Step 1. per-warp partial scan
                                                    // (Performed by all threads in block,
                                                    // with threads in same warp communicating
                                                    // through shared memory buffer ‘ptr’)

   if (lane == 31)  ptr[warp_id] = ptr[idx];        // Step 2. thread 31 in each warp copies
   __syncthreads();                                 // partial-scan bases in per-block
                                                    // shared mem

   if (warp_id == 0) scan_warp(ptr, idx);           // Step 3. scan to accumulate bases
   __syncthreads();                                 // (only performed by warp 0)

   if (warp_id > 0)                                 // Step 4. apply bases to all elements
       val = val + ptr[warp_id-1];                  // (performed by all threads in block)
   __syncthreads();

   ptr[idx] = val;
}
```

Building a larger scan

Example: one million element scan (1024 elements per block)



Exceeding 1 million elements requires partitioning phase two into multiple blocks

Scan implementation

- **Parallelism**
  - Scan algorithm features O(N) parallel work
  - But efficient implementations only leverage as much parallelism as required to make good utilization of the machine
    - Goal is to reduce work and reduce communication/synchronization
- **Locality**
  - Multi-level implementation to match memory hierarchy
    (CUDA example: per-block implementation carried out in local memory)
- **Heterogeneity**: different strategy at different machine levels
  - CUDA example: Different algorithm for intra-warp scan than inter-thread scan
  - Low core count CPU example: based largely on sequential scan

Parallel Segmented Scan

Segmented scan

- Common problem: operating on sequence of sequences
- Examples:
  - For each vertex in a graph:
    - For each edge incoming to vertex:
  - For each particle in simulation
    - For each particle within cutoff radius
- Also there's two levels of parallelism in the problem that a programmer might want to exploit
- But its irregular: the size of edge lists, particle neighbor lists, etc, may be very different from vertex to vertex (or particle to particle)

Segmented scan

- **Generalization of scan**
- **Simultaneously perform scans on arbitrary contiguous partitions of input collection**

```
let A  = [[1,2],[6],[1,2,3,4]]
let ⊕  = +
segmented_scan_exclusive(⊕,A) = [[0,1], [0], [0,1,3,6]]
```

We'll assume a simple "head-flag" representation:

```
A = [[1,2,3],[4,5,6,7,8]]
flag: 0 0 0 1 0 0 0 0
data: 1 2 3 4 5 6 7 8
```

Work-efficient segmented scan (with ⊕ = "+")

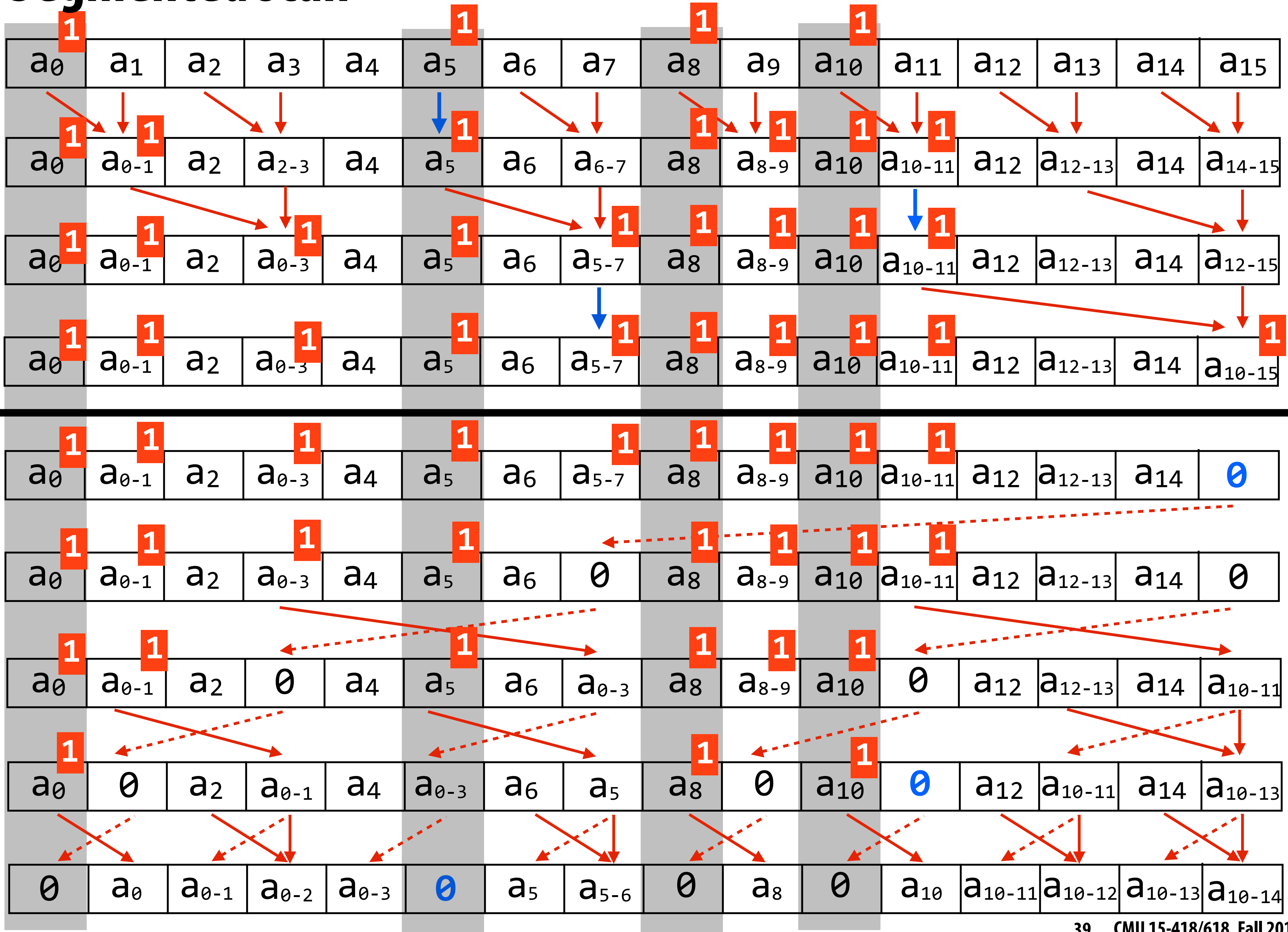
Up-sweep:

```
for d=0 to (log2n - 1) do:
   forall k=0 to n-1 by 2^(d+1) do:
     if flag[k + 2^(d+1) - 1] == 0:
        data[k + 2^(d+1) - 1] = data[k + 2^d - 1] + data[k + 2^(d+1) - 1]
     flag[k + 2^(d+1) - 1] = flag[k + 2^d - 1] || flag[k + 2^(d+1) - 1]
```

Down-sweep:

```
data[n-1] = 0
for d=(log2n - 1) down to 0 do:
   forall k=0 to n-1 by 2^(d+1) do:
     tmp = data[k + 2^d - 1]
     data[k + 2^d - 1] = data[k + 2^(d+1) - 1]
     if flag_original[k + 2^d] == 1:       # must maintain copy of original flags
        data[k + 2^(d+1) - 1] = 0           # start of segment
     else if flag[k + 2^d - 1] == 1:
        data[k + 2^(d+1) - 1] = tmp
     else:
        data[k + 2^(d+1) - 1] = tmp + data[k + 2^(d+1) - 1]
     flag[k + 2^d - 1] = 0
```

Segmented scan



Sparse matrix multiplication example

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 3 & 0 & 1 & \cdots & 0 \\ 0 & 2 & 0 & \cdots & 0 \\ 0 & 0 & 4 & \cdots & 0 \\ & \vdots & & & \\ 0 & 2 & 6 & \cdots & 8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

- **Most values in matrix are zero**
  - **Note: logical parallelization is across per-row dot products**
  - **But different amounts of work per row (complicates wide SIMD execution)**
- **Example sparse storage format: compressed sparse row**

values = [[3,1], [2], [4], ..., [2,6,8]]

cols = [[0,2], [1], [2], ...,]

row_starts = [0, 2, 3, 4, ...]

Sparse matrix multiplication with scan

values = [[3,1], [2], [4], [2,6,8]]

cols = [[0,2], [1], [2], [1,2,3]]

row_starts = [0, 2, 3, 4]

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 6 & 8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

1. Map over all non-zero values: products[i] = values[i] * x[cols[i]]
   - products = [$3x_0$, x_2, $2x_1$, $4x_2$, $2x_1$, $6x_2$, $8x_3$]
2. Create flags vector from row_starts: flags = [1,0,1,1,0,0]
3. Inclusive segmented-scan on (multiples, flags) using addition operator
   - [$3x_0$, $3x_0+x_2$, $2x_1$, $4x_2$, $2x_1$, $2x_1+6x_2$, $2x_1+6x_2+8x_2$]
4. Take last element in each segment:
   - y = [$3x_0+x_2$, $2x_1$, $4x_2$, $2x_1+6x_2+8x_2$]

Scan/segmented scan summary

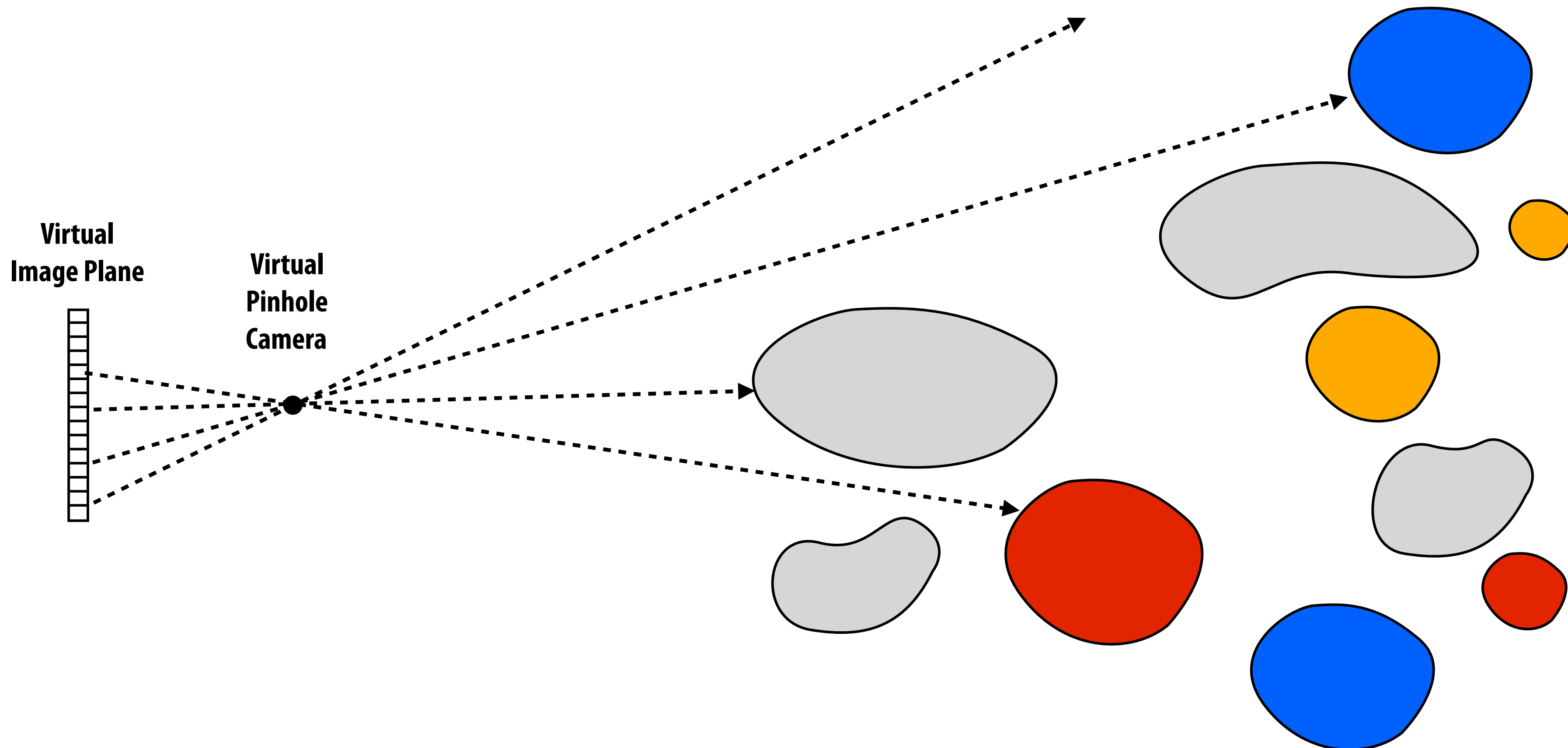
- **Scan**
  - Parallel implementation of (intuitively sequential application)
  - Theory: parallelism linear in number of elements
  - Practice: exploit locality, use only as much parallelism as necessary to fill the machine
    - Great example of applying different strategies at different levels of the machine
- **Segmented scan**
  - Express computation and operate on irregular data structures (e.g., list of lists) in a regular, data parallel way

Parallel Ray Tracing on SIMD Architectures

(since many students always ask about parallel ray tracing)

Ray tracing

Problem statement:
Given a "ray", find closest intersection with scene geometry



Simplest ray tracer:

For each image pixel, shoot ray from camera through pixel into scene.

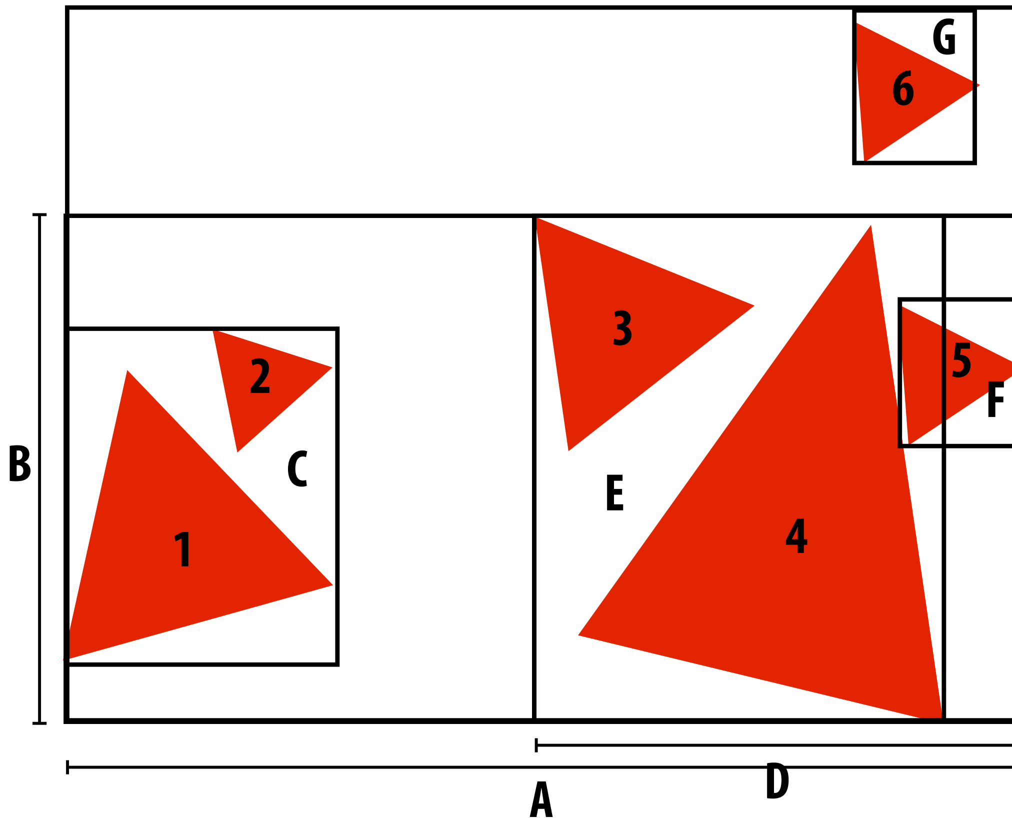
Color pixel according to first surface hit.

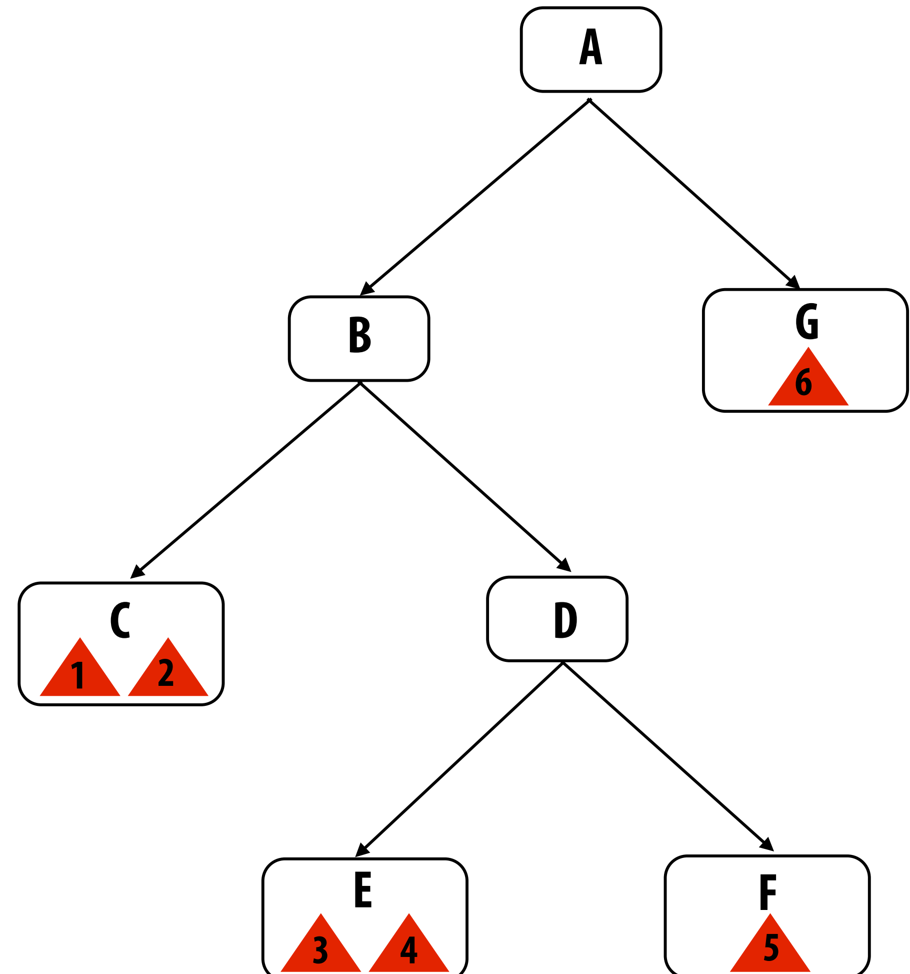
Accelerating ray-scene intersection

Preprocess scene to build data structure that accelerates finding "closest" geometry along ray

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

- **Hierarchical grouping adapts to non-uniform density of scene objects**





Parallelize across rays

- Simultaneously intersect multiple rays with scene

- Different cores trace different rays in parallel
  - Trivial "embarrassingly parallel" implementation

- But how to leverage SIMD parallelism within a core?

- Today: we'll discuss one approach: ray packets
  - Code is explicitly written to trace N rays at a time, not 1 ray

Simple ray tracer (using a 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);
   }
}
```

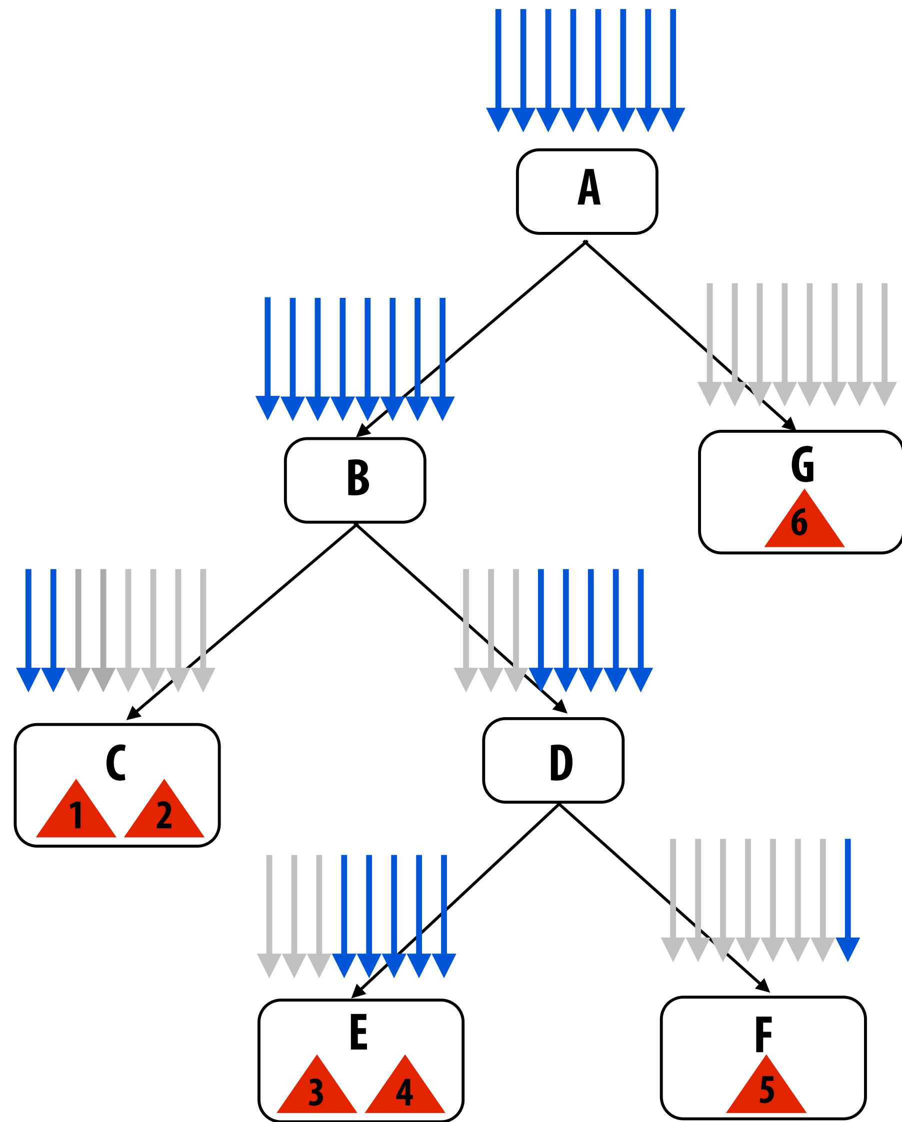
Ray packet tracing

[Wald et al. 2001]

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



Note: r6 does not pass node F box test due to closest-so-far check, and thus does not visit F

Advantages of packets

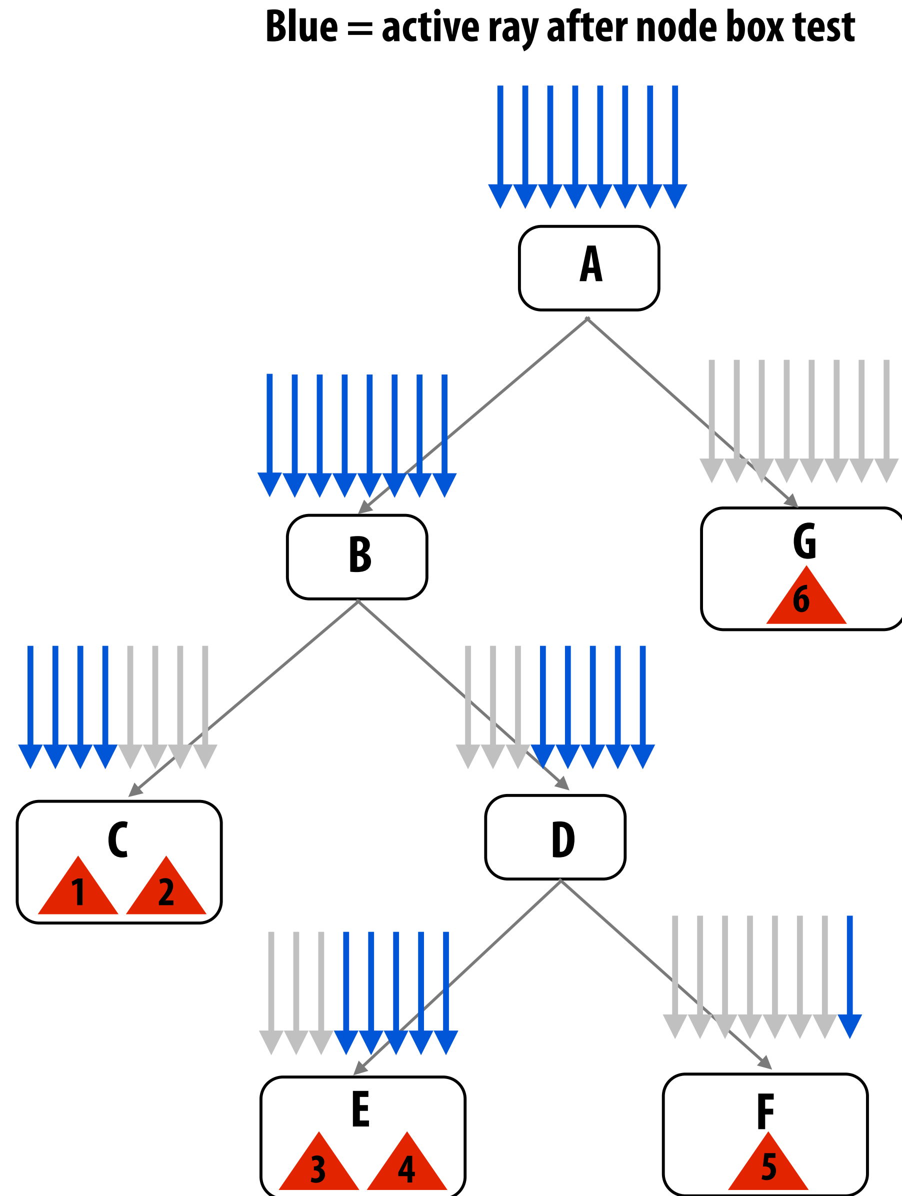
- Map packet operations to wide SIMD execution
  - One vector lane per ray
- Amortize BVH data fetch: all rays in packet visit node at same time
  - Load BVH node once for all rays in packet (not once per ray)
  - Note: there is value to making packets bigger than SIMD width! (e.g., size = 64)
- Amortize work (packets are hierarchies over rays)
  - Use interval arithmetic to conservatively test entire set of rays against node bbox (e.g., think of a packet as a beam)
  - Further arithmetic optimizations possible when all rays share origin
  - Note: there is 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)

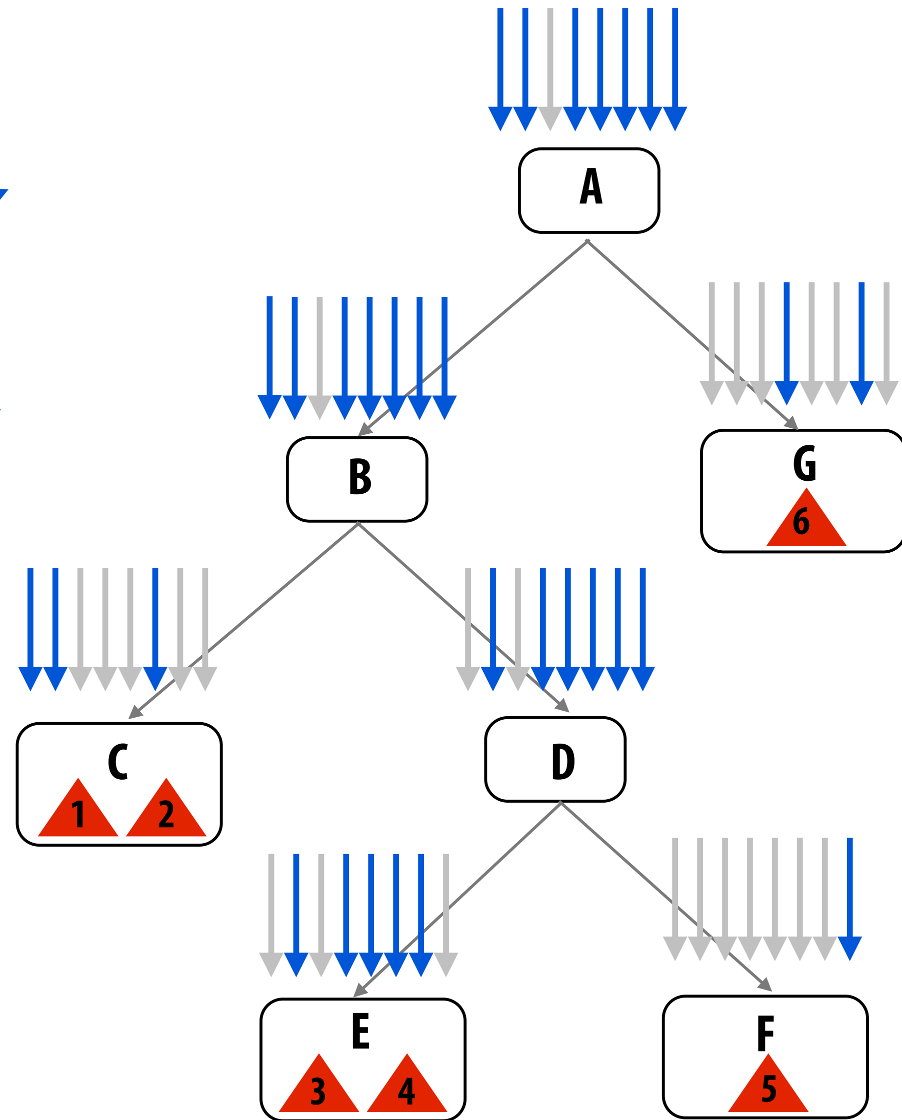
- Loss of efficiency: node traversal, intersection, etc. amortized over less than a packet's worth of rays

- Not all SIMD lanes doing useful work



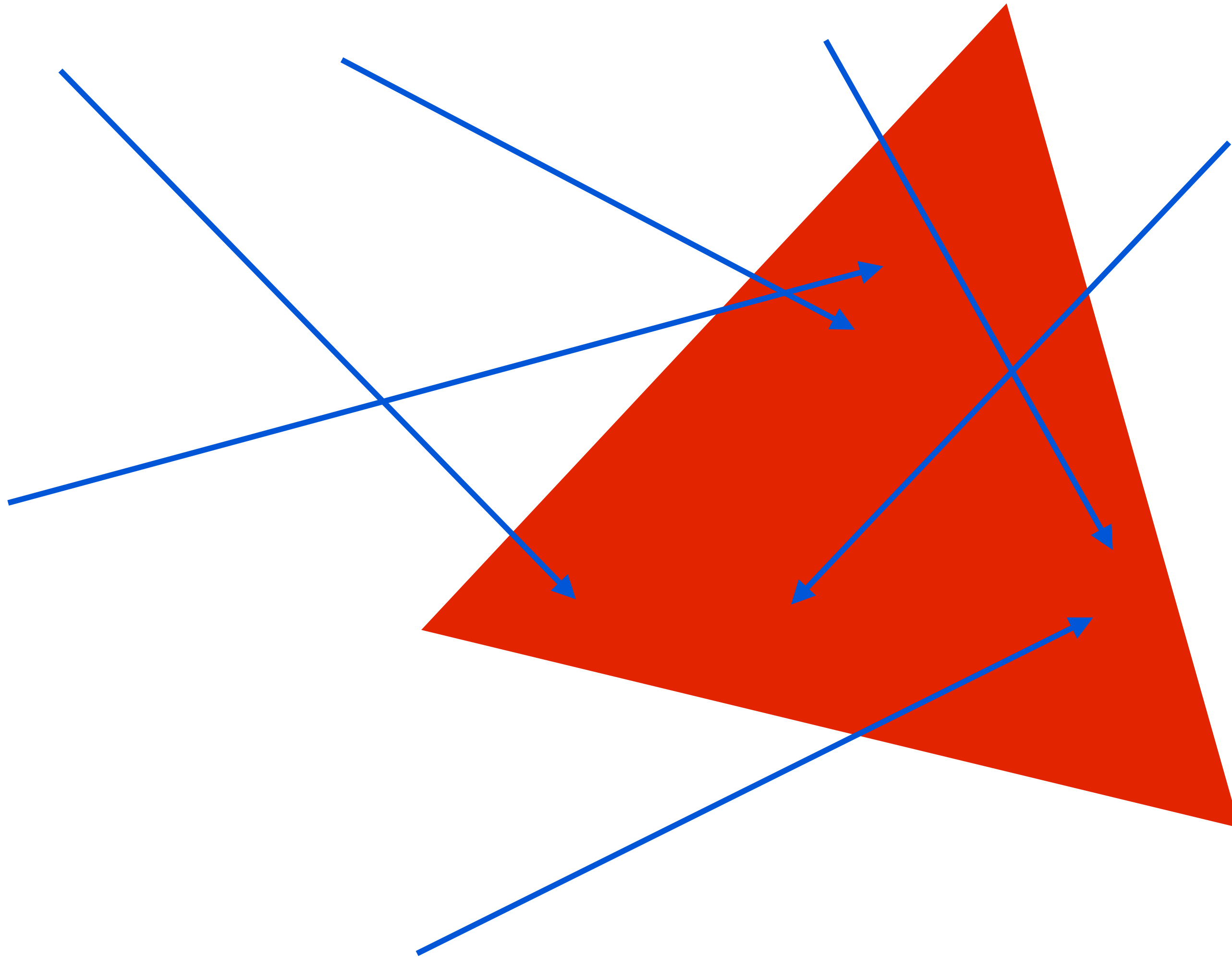
Ray packet tracing: incoherent rays

Blue = active ray after node box test



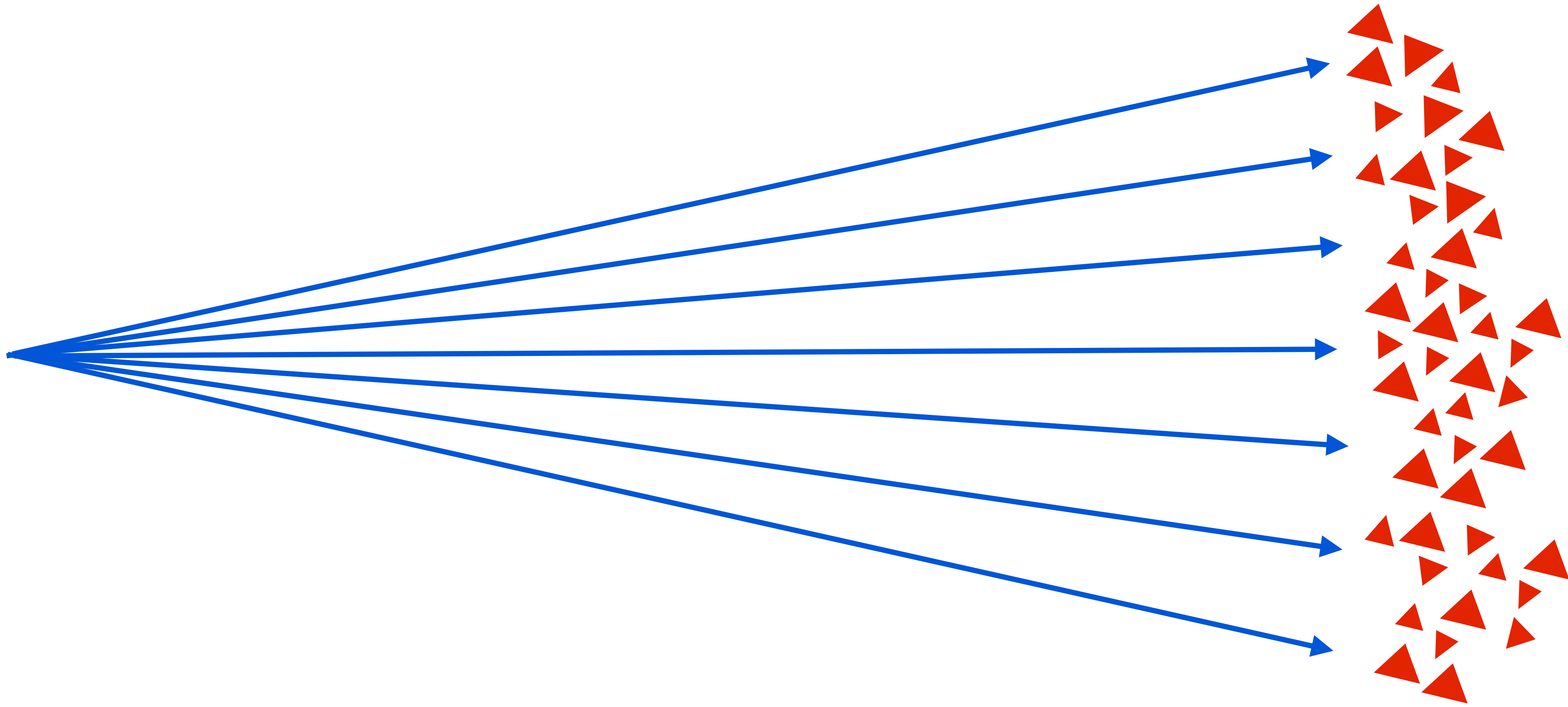
When rays are incoherent, benefit of packets can decrease significantly. This example: packet visits all tree nodes. (So all eight rays visit all tree nodes! No culling benefit!)

Incoherence is a property of <u>both</u> the rays and the scene



Random rays are "coherent" with respect to the BVH if the scene is one big triangle!

Incoherence is a property of both the rays and the scene



Camera rays become "incoherent" with respect to lower nodes in the BVH if a scene is overly detailed

(Side note: this suggests the importance of choosing the right geometric level of detail)

Improving packet tracing with ray reordering

[Boulos et al. 2008]

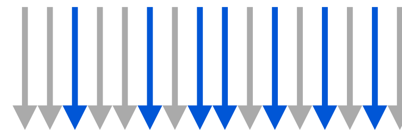
Idea: when packet utilization drops below threshold, resort rays and continue with smaller packet

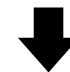
- Increases SIMD utilization
- Amortization benefits of smaller packets, but not large packets

---

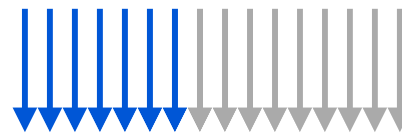
Example: consider 8-wide SIMD processor and 16-ray packets (2 SIMD instructions required to perform each operation on all rays in packet)

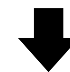
16-ray packet: 7 of 16 rays active



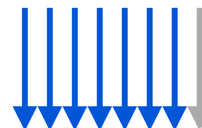


Reorder rays
Recompute intervals/bounds for active rays





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



Giving up on packets

- Even with reordering, ray coherence during BVH traversal will diminish
  - Diffuse bounces result in essentially random ray distribution
  - High-resolution geometry encourages incoherence near leaves of tree

- In these situations there is little benefit to packets (can even decrease performance compared to single ray code)

Packet tracing best practices

- Use large packets for eye/reflection/point light shadow rays or higher levels of BVH [Wald et al. 2007]
  - Ray coherence always high at the top of the tree

- Switch to single ray (intra-ray SIMD) when packet utilization drops below threshold [Benthin et al. 2011]
  - For wide SIMD machine, a branching-factor-4 BVH works well for both packet traversal and single ray traversal

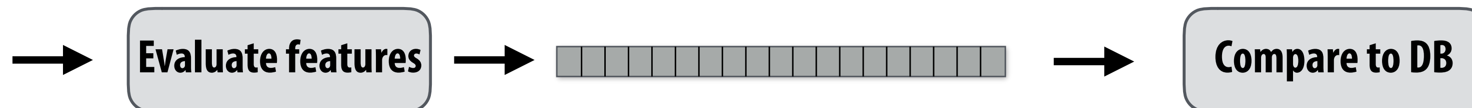
- Can use packet reordering to postpone time of switch [Boulos et al. 2008]
  - Reordering allows packets to provide benefit deeper into tree
  - Not often used in practice due to high implementation complexity

Summary

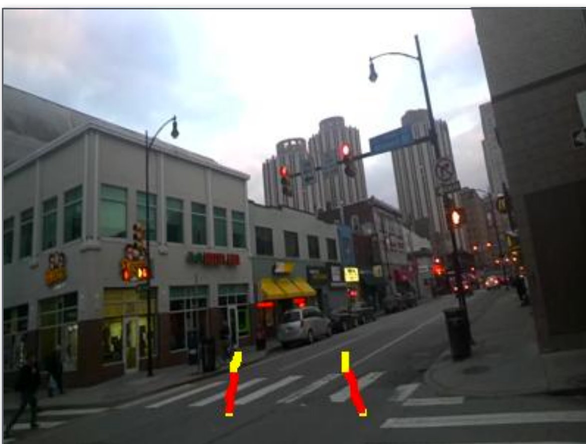
- Today we looked at several different parallel programs
- Key questions:
  - What are the dependencies?
  - What synchronization is necessary?
  - How to balance the work?
  - How to exploit locality inherent in the problem?
- Trends
  - Only need enough parallelism to keep all processing elements busy (e.g., data-parallel scan vs. simple multi-core scan)
  - Different parallelization strategies may be applicable under different workloads (packets vs. no packets) or different locations in the machine (different implementations of scan internal and external to warp)

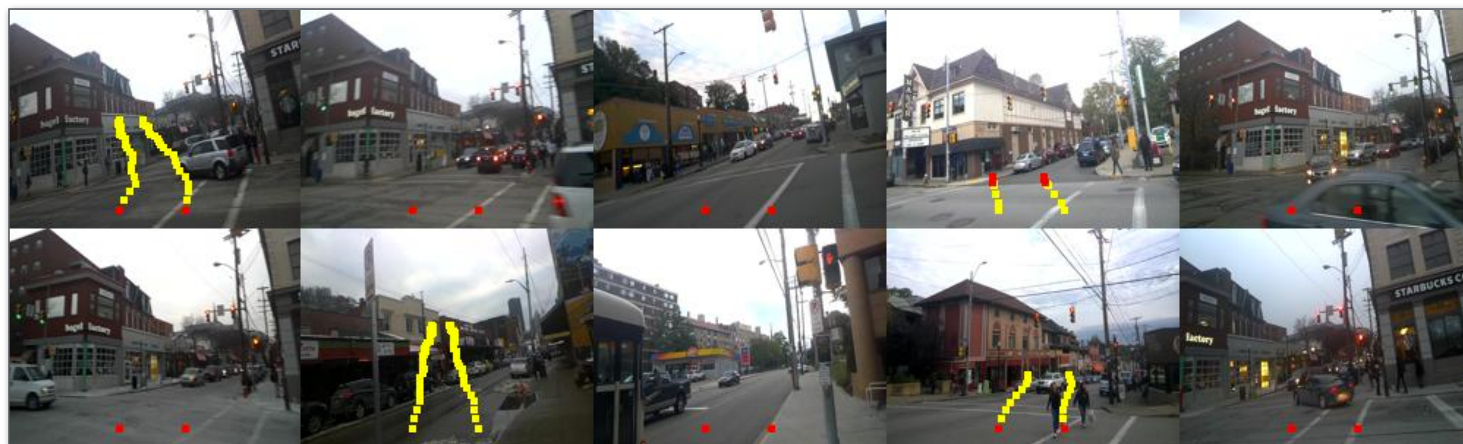
Brute force nearest neighbor search

Task: find most similar images to query



Query image **Similar images from database**





* Trajectory shown on image is movement of camera, not part of image

Find most similar images to a query

Brute force computation:

```
#define NUM_DB_IMAGES 50000
#define FEATURE_LEN   9216

float compute_distances(float* a, float* b) {
    float result = 0.f;
    for (int i=0; i<FEATURE_LEN; i++)
       result += a[i] * b[i];
    return result;
}

float features[NUM_DB_IMAGES][FEATURE_LEN];   // ~1.7 GB

forall queries i:
   forall images j in database:
       results[i][j] = compute_distance(features[i], features[j]);
```

Batching queries

Increase arithmetic intensity: amortize load of DB data across eight queries

```
#define NUM_DB_IMAGES 50000
#define FEATURE_LEN   9216

float compute_distances_batched(float* results, float* a, float* b) {
    for (int ii=0; ii<8; ii++)
       results[i+ii] = 0.0
    for (int i=0; i<FEATURE_LEN; i++)
       for (int ii=0; ii<8; ii++)
          results[ii] += a[(ii*FEATURE_LEN)+i] * b[i];
}

float features[NUM_DB_IMAGES][FEATURE_LEN];   // ~1.7 GB

forall queries i (by 8):
   forall images j in database:
       for (int ii=0; ii<8; ii++)
         compute_distance_batched(results[i], features[i+ii], features[j]);
```

Performance