



Guess who's throwing the dice? Yeeaahhh!

Lecture 24:

Fun With Parallel Algorithms

Segmented Scan

Neutral territory methods

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

Announcements

- **Checkpoint writeup due Monday April 23rd, 11:59pm**
 - Which should be trivial if you have kept a log of activities, rants, thinking, findings, on your project web page
 - It will be interesting for us to read
 - It will come in handy when it comes time to do your writeup
 - Writing clarifies thinking

- **Exam 2 is a week from Thursday**
 - Kayvon's exam prep office hours: Sunday 8pm-11pm
 - TA review session: Tuesday, time TBD (in class or Tuesday evening)

Assignment 4 solutions

- **Good solutions picked a master processor to direct slaves**
- **Often master did not participate in real computation**

- **Reminder:**

- **Get code working**
- **Measure key performance properties**
- **Diagnose issues**
- **Optimize**



Repeat

Today

- **Fun algorithms for lots of processors!**
 - **Review of scan, implementing scan**
 - **Generalization to segmented scan**
 - **Neutral territory methods**

Review: scan

let $a = [a_0, a_1, a_2, a_3, \dots, a_{n-1}]$

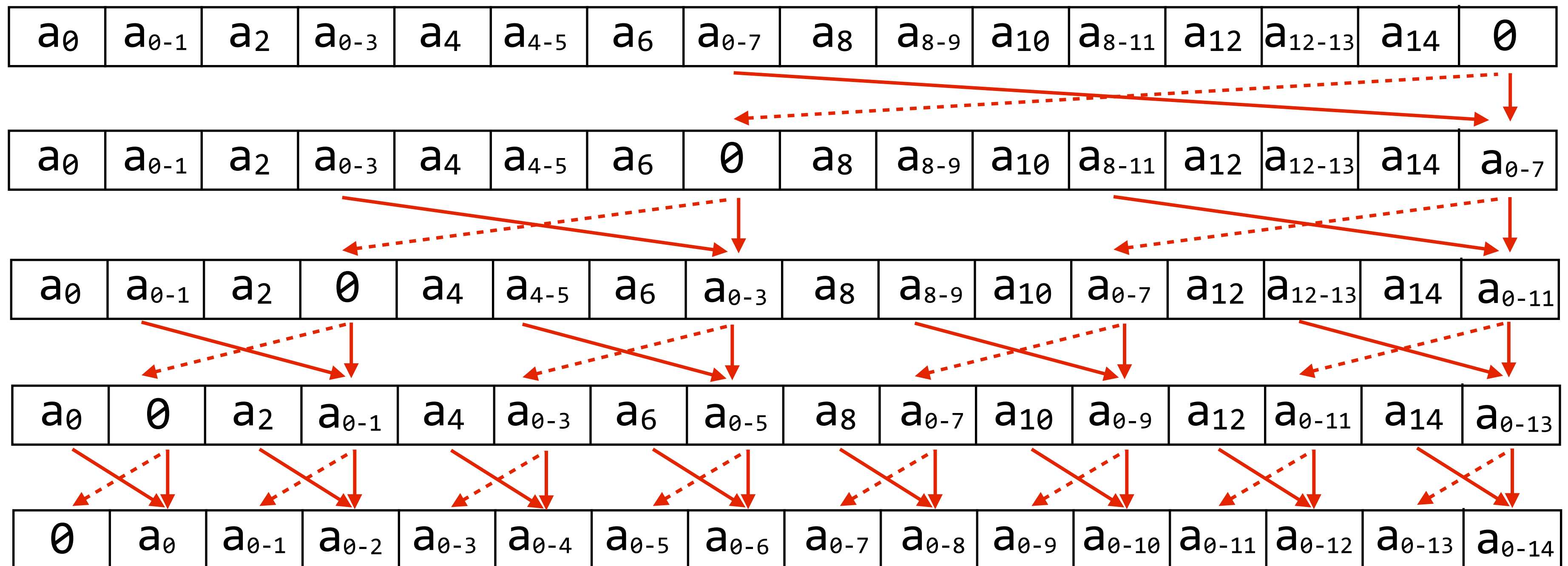
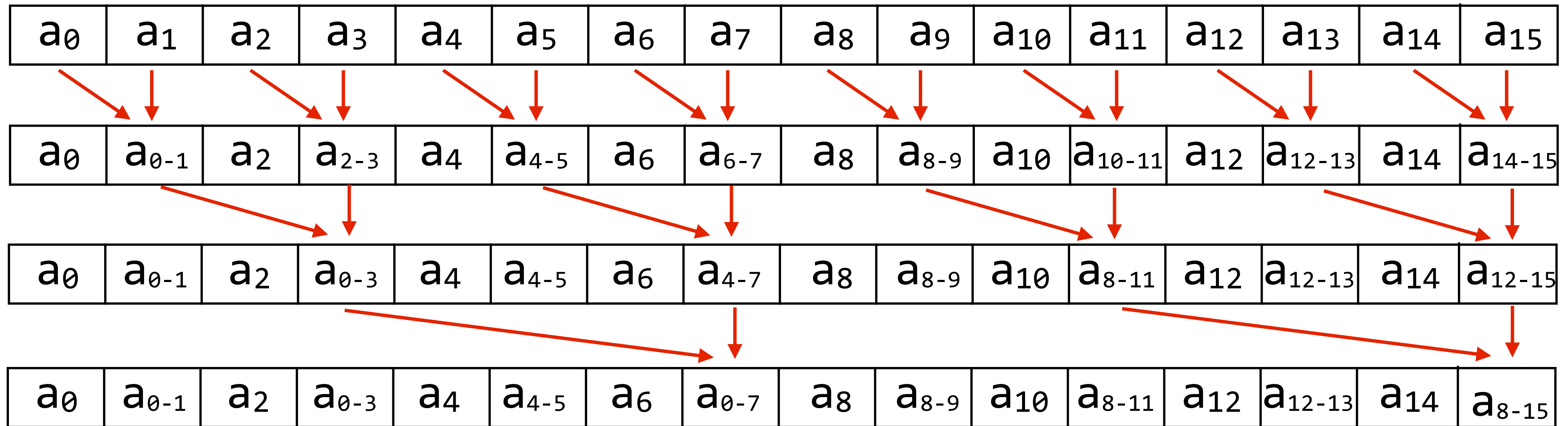
let \oplus **be an associative binary operator with identity element** I

scan_inclusive $(\oplus, a) = [a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \dots$

scan_exclusive $(\oplus, a) = [I, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \dots$

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

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



Exclusive scan algorithm

Up-sweep:

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

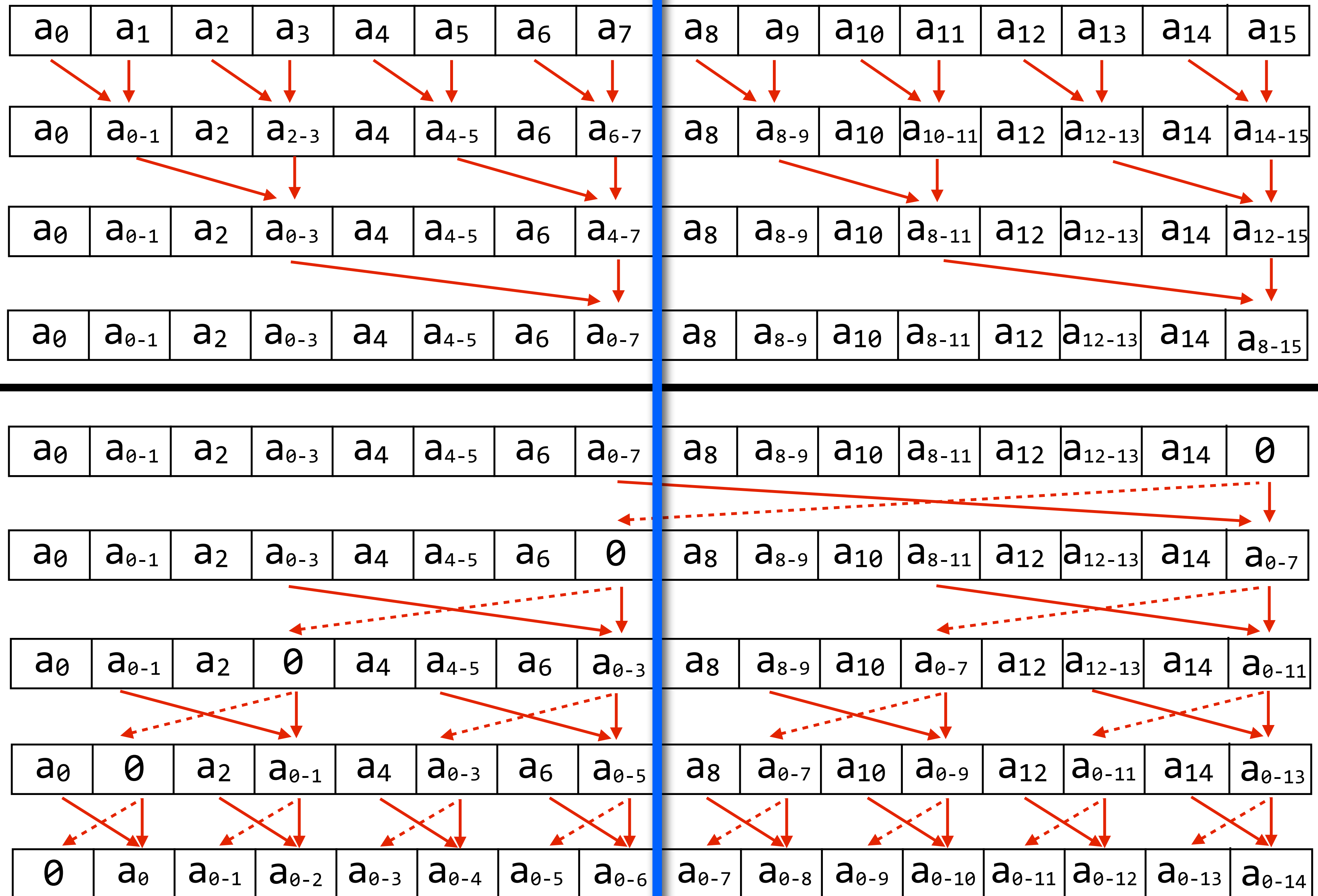
Down-sweep:

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

Work: $O(N)$ (what is the constant?)

Locality: ??

Implement scan: two cores



In practice: multi-core implementation

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}
-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	----------	----------	----------	----------	----------	----------

P1

Sequential scan on elements [0-7]

Let base = $(a_0 + \dots + a_7)$

(exclusive scan result for last element)

Add base to elements a_8 thru a_{15}

P2

Sequential scan on elements [8-15]

Add base to elements a_{12} thru a_{15}

SIMD implementation

Example: **exclusive scan 32-element array**

32-wide GPU execution (SPMD program)

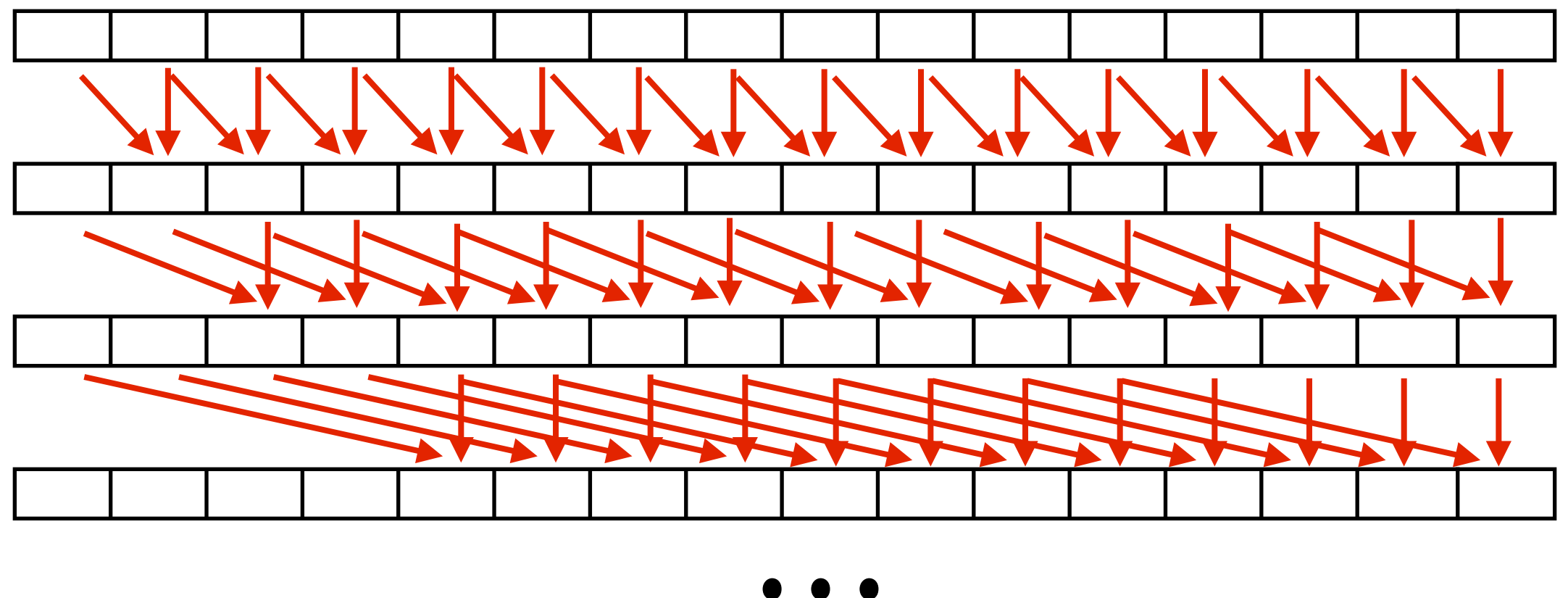
Exclusive result is returned (note: state of `ptr[]` is inclusive result) **CUDA thread index**

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

    if (lane >= 1) ptr[idx] = OP::apply(ptr[idx - 1], ptr[idx]);
    if (lane >= 2) ptr[idx] = OP::apply(ptr[idx - 2], ptr[idx]);
    if (lane >= 4) ptr[idx] = OP::apply(ptr[idx - 4], ptr[idx]);
    if (lane >= 8) ptr[idx] = OP::apply(ptr[idx - 8], ptr[idx]);
    if (lane >= 16) ptr[idx] = OP::apply(ptr[idx - 16], ptr[idx]);

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

Work: ??



SIMD implementation

Example: exclusive scan 32-element array

32-wide GPU execution (SPMD program)

CUDA thread index



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

    if (lane >= 1) ptr[idx] = OP::apply(ptr[idx - 1], ptr[idx]);
    if (lane >= 2) ptr[idx] = OP::apply(ptr[idx - 2], ptr[idx]);
    if (lane >= 4) ptr[idx] = OP::apply(ptr[idx - 4], ptr[idx]);
    if (lane >= 8) ptr[idx] = OP::apply(ptr[idx - 8], ptr[idx]);
    if (lane >= 16) ptr[idx] = OP::apply(ptr[idx - 16], ptr[idx]);

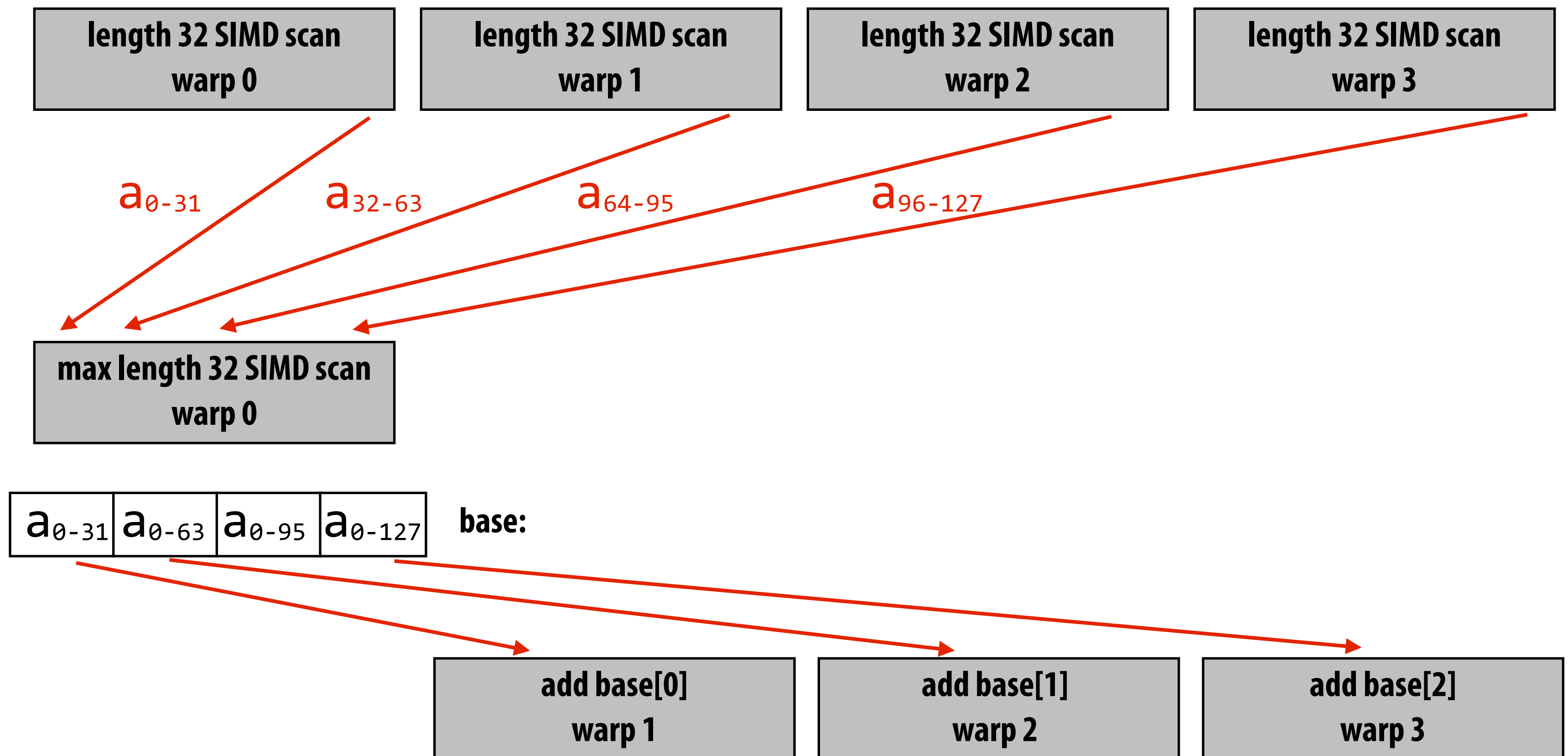
    return (lane > 0) ? ptr[idx - 1] : OP::identity();
}
```

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 a larger scan

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



Multi-threaded, SIMD implementation

Example: cooperating threads in a CUDA thread block

(We provided similar code in assignment 2, assumes length of array given by ptr is same as number of threads per block)

```
template<class OP, class T>
__device__ void scan_block(volatile T *ptr, const unsigned int idx)
{
    const unsigned int lane = idx & 31; // index of thread in warp (0..31)
    const unsigned int warpid = idx >> 5;

    T val = scan_warp<OP,T>(ptr, idx);           // Step 1. per-warp partial scan

    if (lane == 31) ptr[warpid] = ptr[idx];      // Step 2. copy partial-scan bases
    __syncthreads();

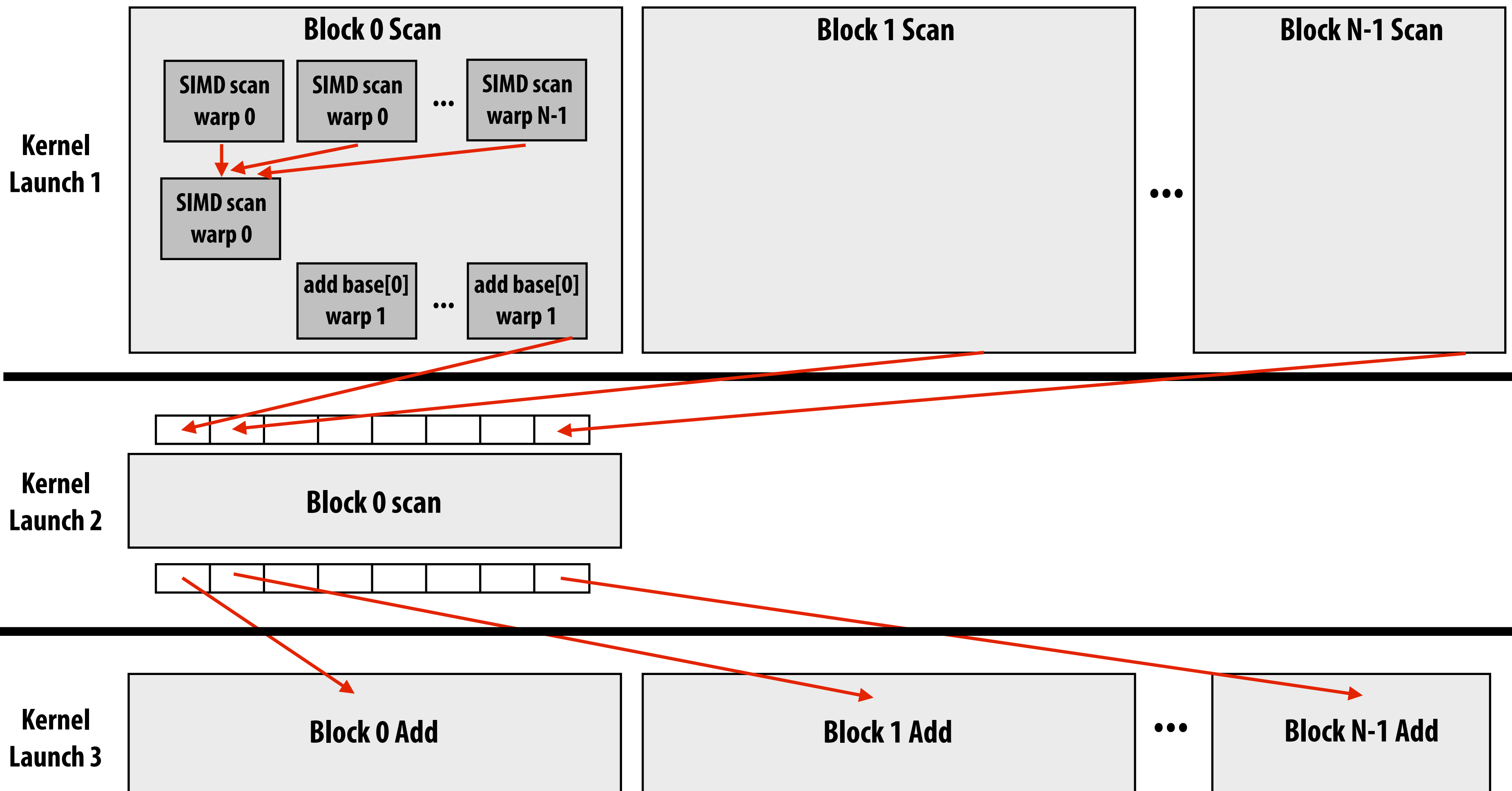
    if (warpid == 0) scan_warp<OP, T>(ptr, idx); // Step 3. scan to accumulate bases
    __syncthreads();

    if (warpid > 0)                               // Step 4. apply bases to all elements
        val = OP::apply(ptr[warpid-1], val);
    __syncthreads();

    ptr[idx] = val;
}
```

Building a larger scan

Example: up 1M element scan (1024 element blocks)



Exceeding 1M elements requires parallelization of phase 2

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
 - Reduce work and reduce communication/synchronization

■ Locality

- Multi-level implementation matches memory hierarchy
(Per-block implementation carried out in local memory)

■ Heterogeneity: different strategy at different machine levels

- Different algorithm for intra-warp scan

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  $\oplus$  = +
```

```
segmented_scan_exclusive( $\oplus$ ,a) = [[0,1], [0], [0,1,3,6]]
```

Assume 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

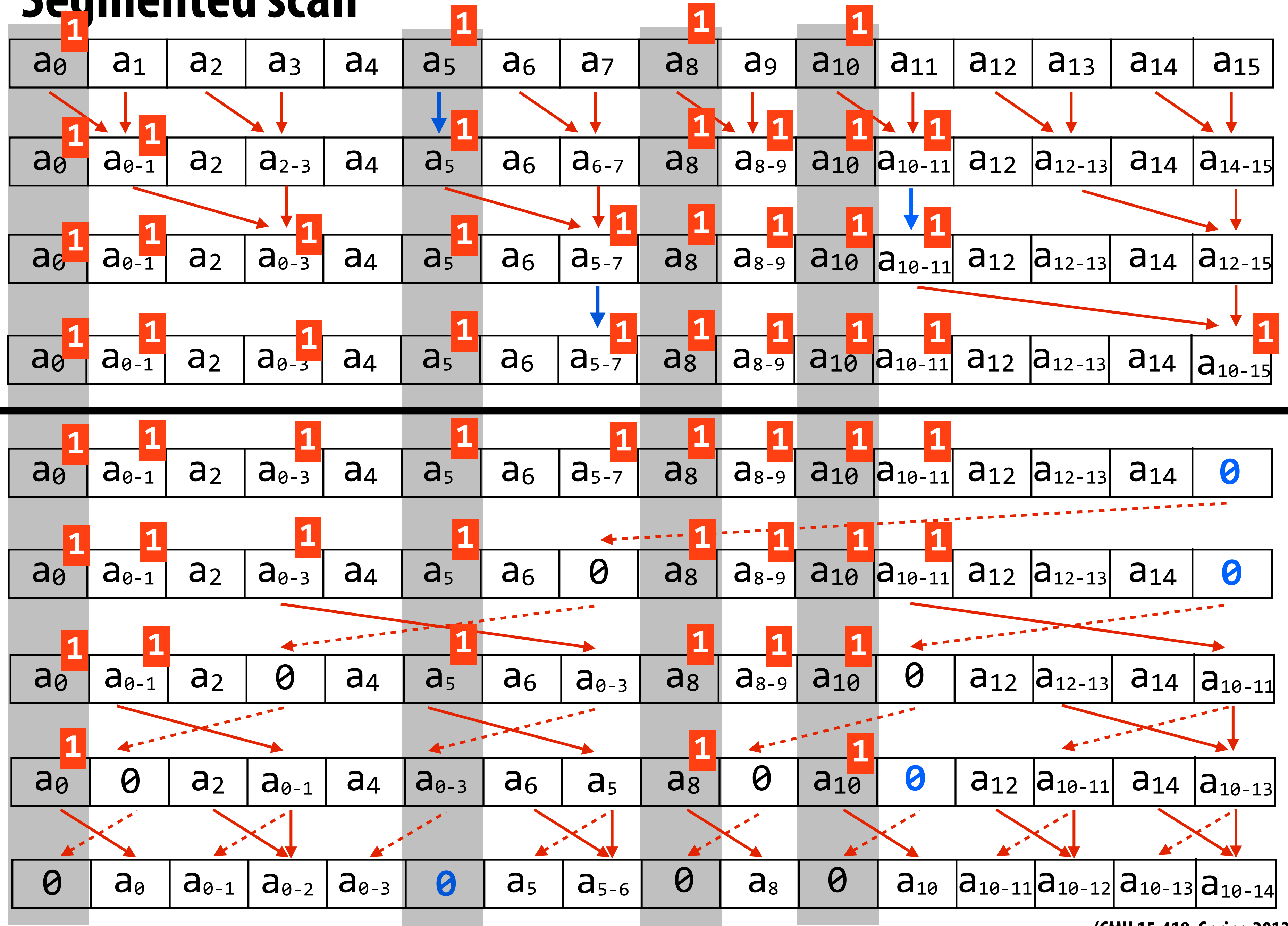
Up-sweep:

```
for d=0 to ( $\log_2 n - 1$ ) do
  forall k=0 to n-1 by  $2^{d+1}$  do
    if flag[k +  $2^{d+1} - 1$ ] == 0:
      data[k +  $2^{d+1} - 1$ ]  $\leftarrow$  data[k +  $2^d - 1$ ] + data[k +  $2^{d+1} - 1$ ]
    flag[k +  $2^{d+1} - 1$ ]  $\leftarrow$  flag[k +  $2^d - 1$ ] || flag[k +  $2^{d+1} - 1$ ]
```

Down-sweep:

```
data[n-1]  $\leftarrow$  0
for d=( $\log_2 n - 1$ ) down to 0 do
  forall k=0 to n-1 by  $2^{d+1}$  do
    tmp  $\leftarrow$  data[k +  $2^d - 1$ ]
    data[k +  $2^d - 1$ ]  $\leftarrow$  data[k +  $2^{d+1} - 1$ ]
    if flag_original[k +  $2^d$ ] == 1: // maintain copy of original flags
      data[k +  $2^{d+1} - 1$ ]  $\leftarrow$  0
    else if flag[k +  $2^d - 1$ ] == 1:
      data[k +  $2^{d+1} - 1$ ]  $\leftarrow$  tmp
    else:
      data[k +  $2^{d+1} - 1$ ]  $\leftarrow$  tmp + data[k +  $2^{d+1} - 1$ ]
    flag[k +  $2^d - 1$ ]  $\leftarrow$  0
```

Segmented scan



Sparse matrix multiplication example

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

- **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]]
row_starts = [0, 2, 3, 4, ...]
cols = [[0,2], [1], [2], ...,]

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-zeros: multiples[i] = values[i] * x[cols[i]]
 - multiples = [3x₀, x₂, 2x₁, 4x₂, 2x₁, 6x₂, 8x₃]
2. Create flags vector from row_starts: flags = [1,0,1,1,0,0]
3. Inclusive segmented-scan on (multiples, flags) using addition operator
 - [3x₀, 3x₀+x₂, 2x₁, 4x₂, 2x₁, 2x₁+6x₂, 2x₁+6x₂+8x₂]
4. Take last element in each segment:
 - y = [3x₀+x₂, 4x₂, 2x₁, 2x₁+6x₂+8x₂]

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**

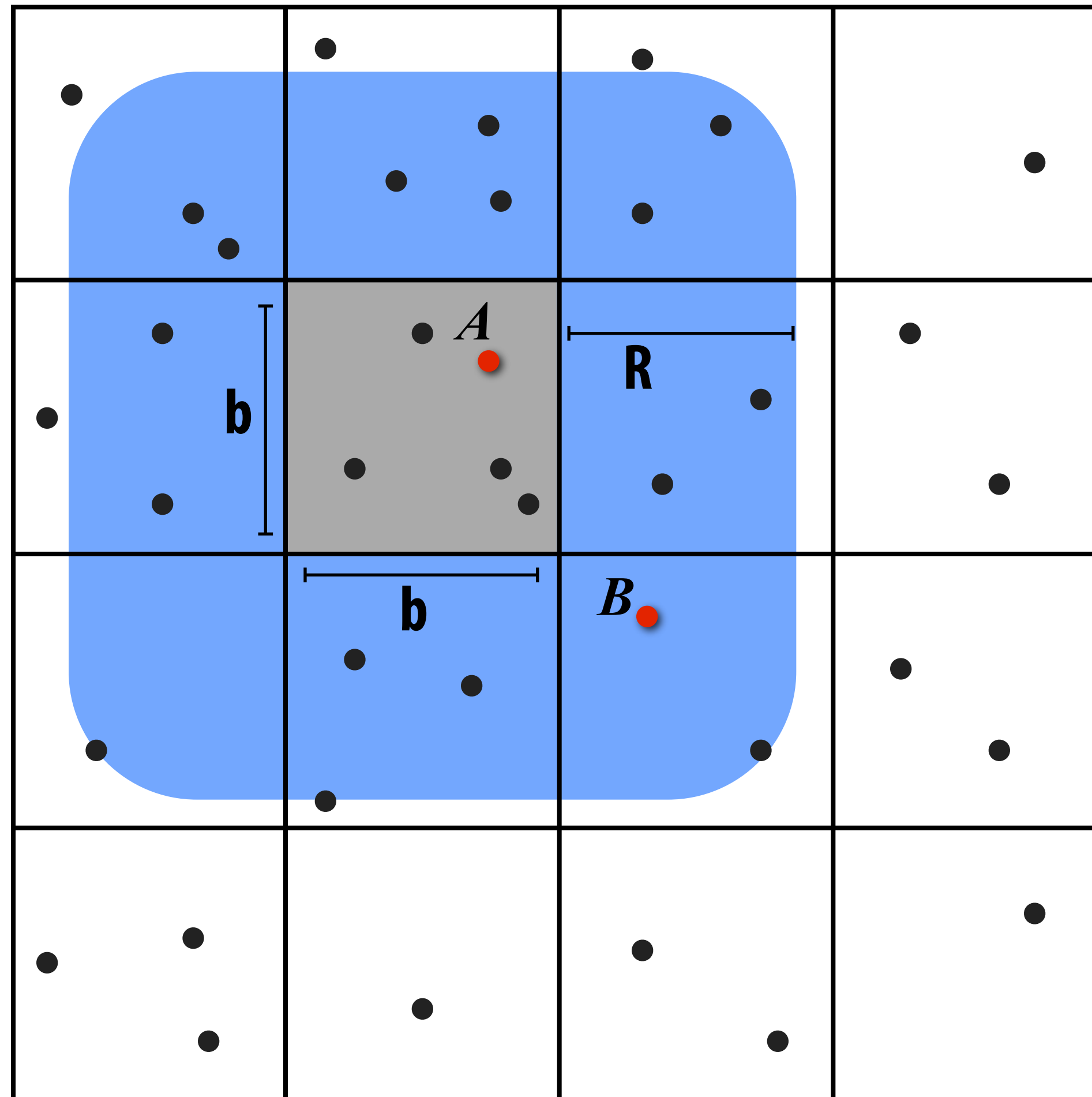
■ Segmented scan

- **Express computation and operate on irregular data structures (e.g., list of lists) in a regular, data parallel way**

Neutral Territory Method

Limited range force computation

Goal: compute interactions between all particles located within distance R



Partition space into P regions (one region per processor)

Two sets of particles:

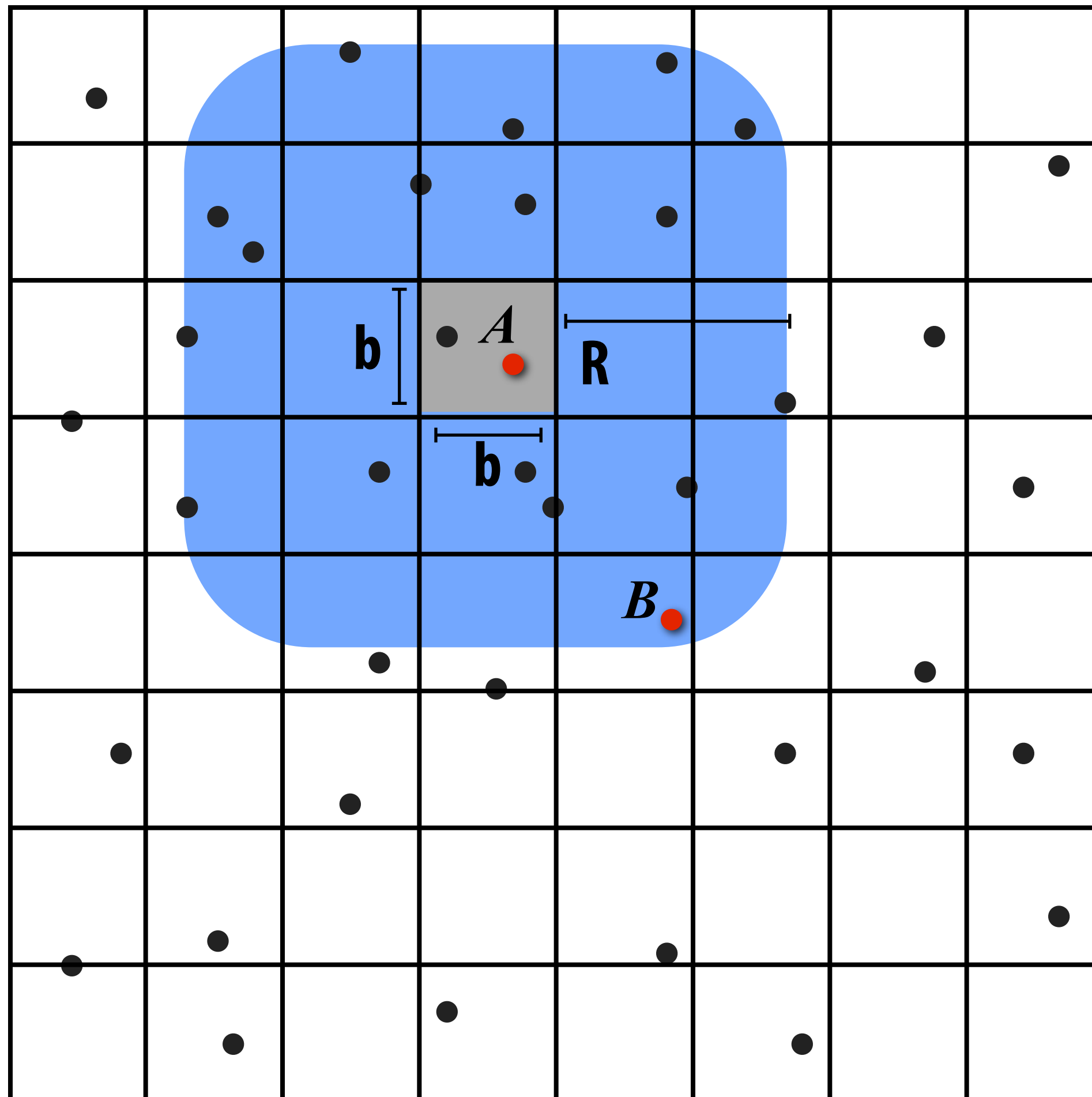
Home region: $\sim b^2$
(particles in gray box, “owned” by processor P)

Import region: $\sim 4bR + \pi R^2$
(particles that must be communicated to processor P to perform computation)

Number of interactions carried out by processor P is proportional to product of the two terms:
 $\sim b^2(4bR + \pi R^2)$

Limited range force computation: scaling

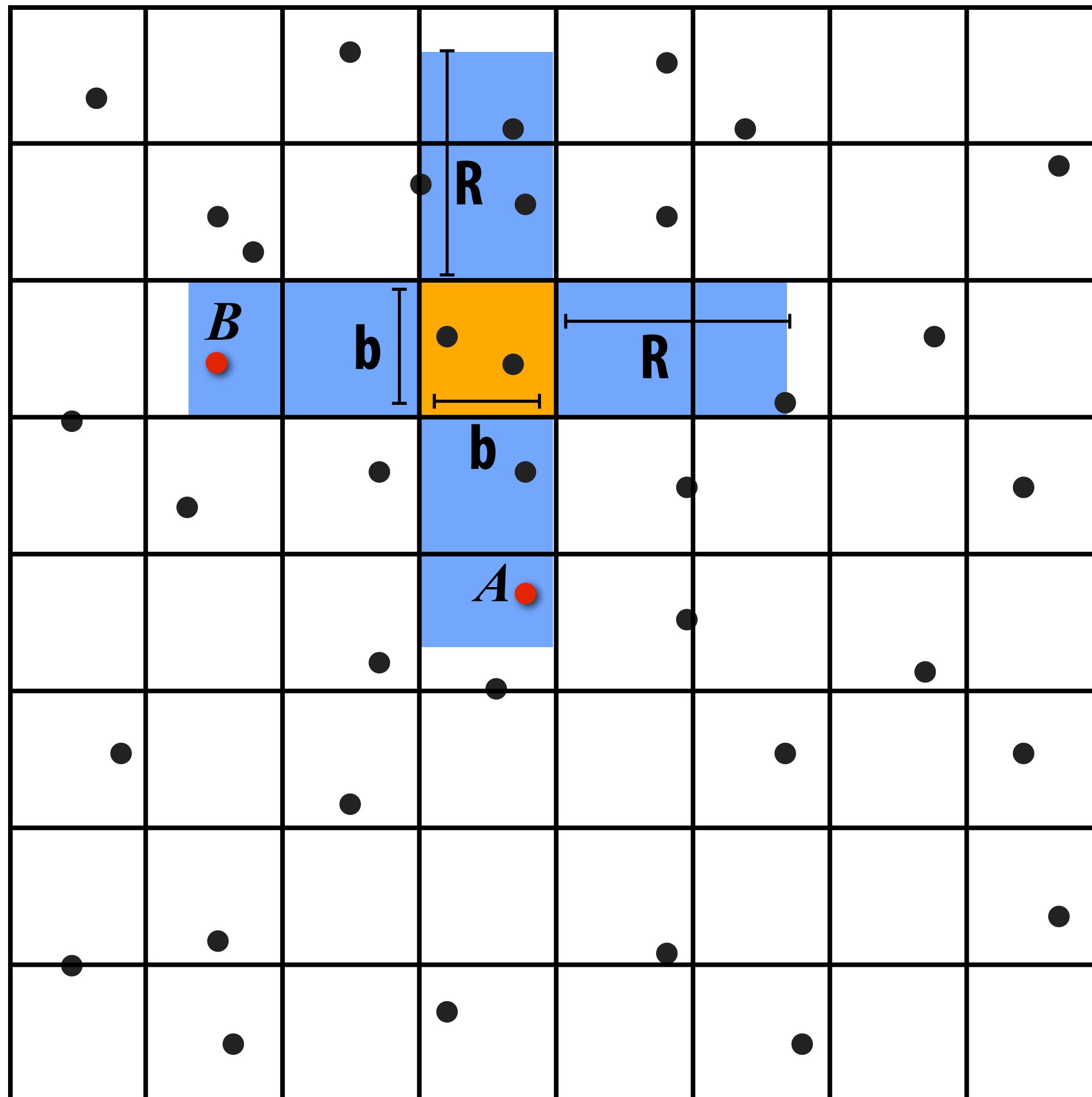
Goal: compute interactions between all particles located within distance R



Consider comm. to comp. ratio as $P \rightarrow \infty$:
 $b \sim 1/\sqrt{P}$

Import region shrinks to πR^2
Home region shrinks to 0

“Neutral territory” (NT) approach (2D)



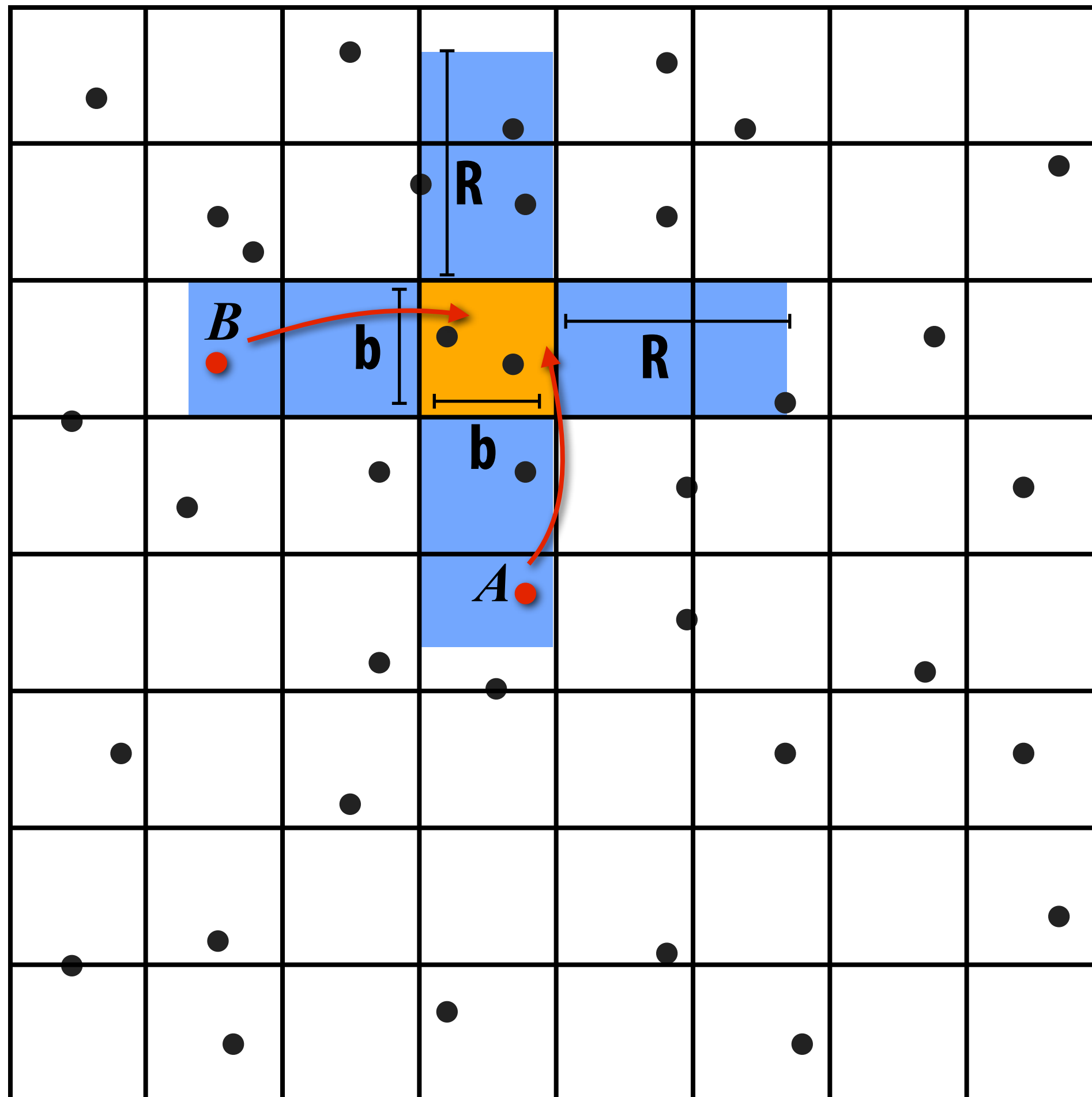
Particle A interacts with particle B in square with x coord equal to that of A and y coord equal to that of B.

Import region of the yellow square associated with processor P is highlighted ROW and highlighted COLUMN.

Size of import region: $4bR$

Size of import region as $P \rightarrow \infty$: 0

“Neutral territory” intuition



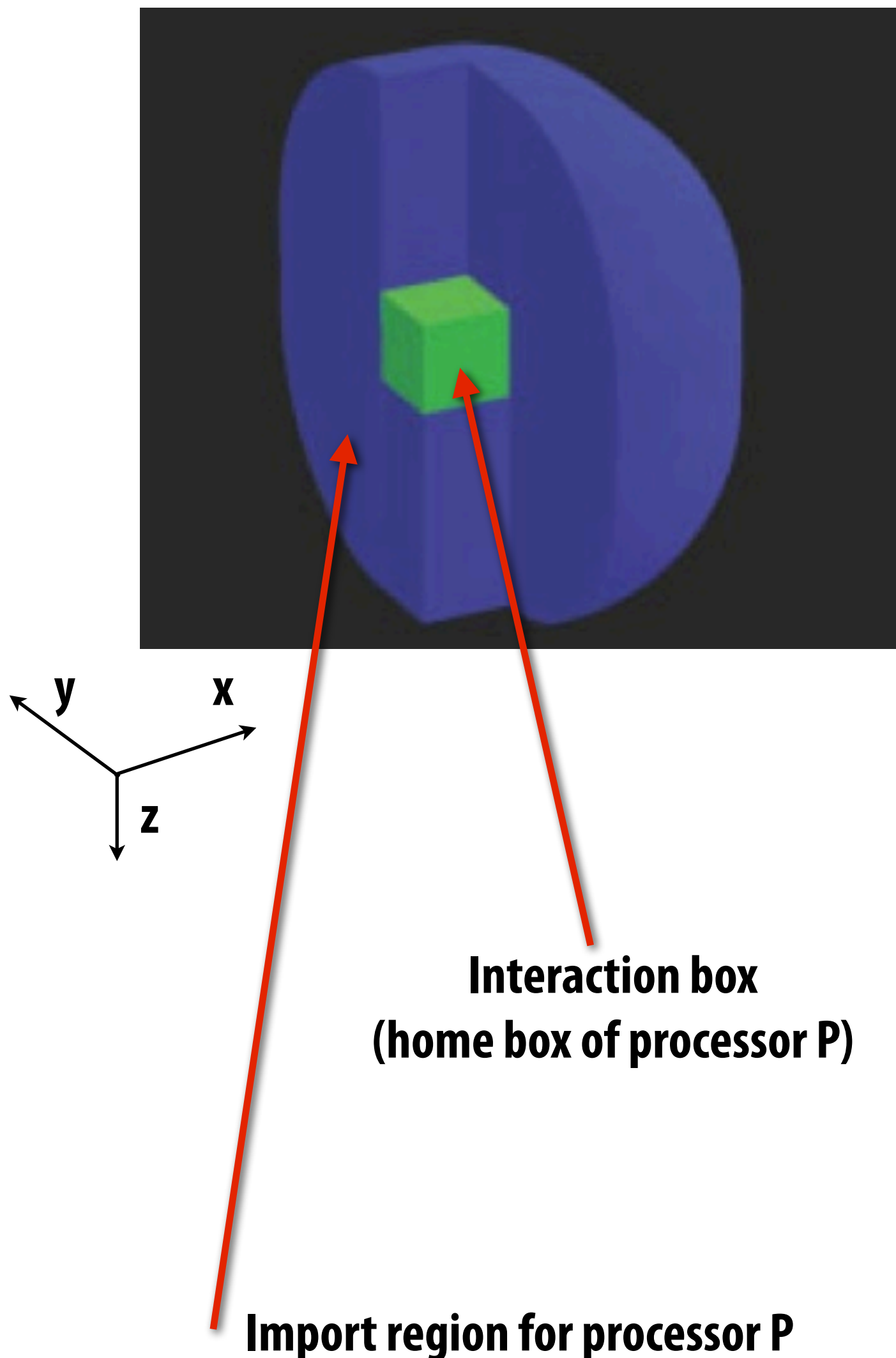
Assuming $R > b$ (true for high processor count), most particle interactions computed by processor upon which neither particle resides!

Intuition: NT method two sets of interacting particles (column and row) are the same size. Recall number of interactions is the product of these sizes.

Analogy: perimeter of a square is greater than any other quadrilateral with the same area.

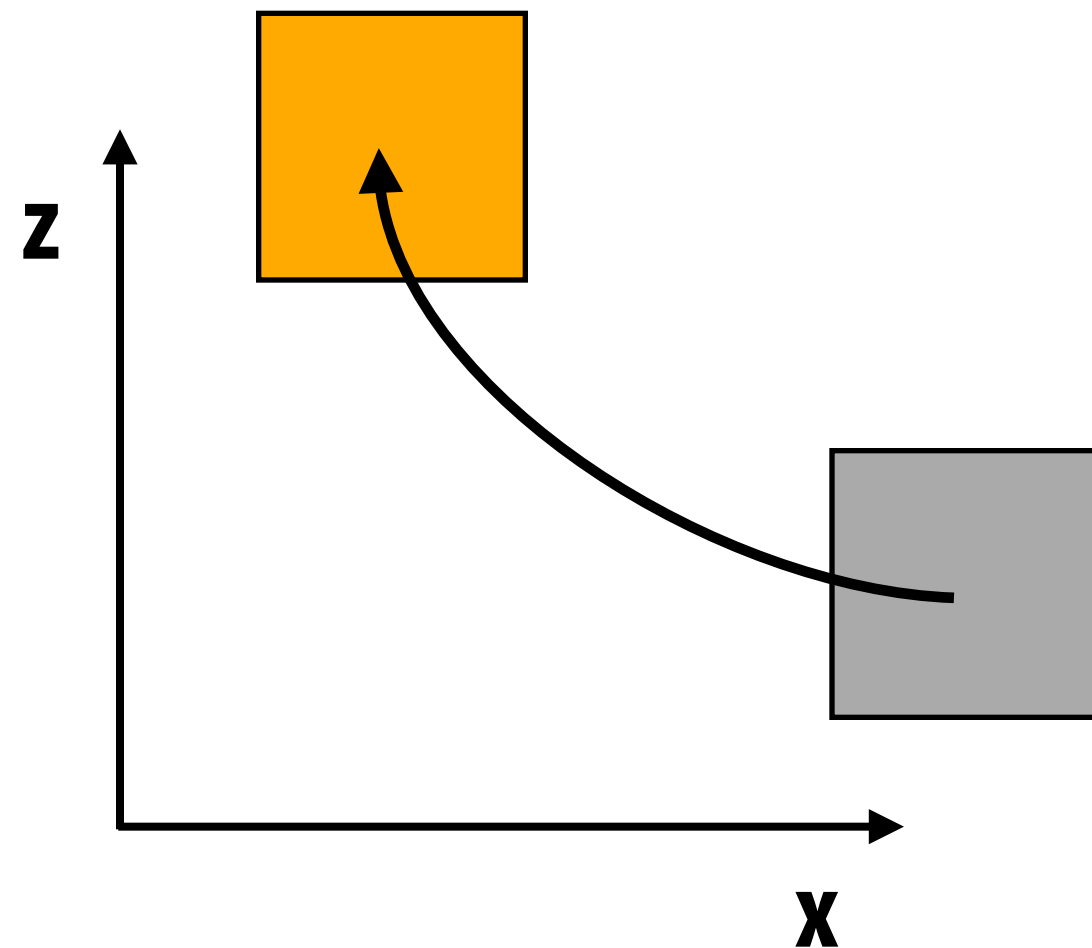
Extending to 3D

"Half-shell" method



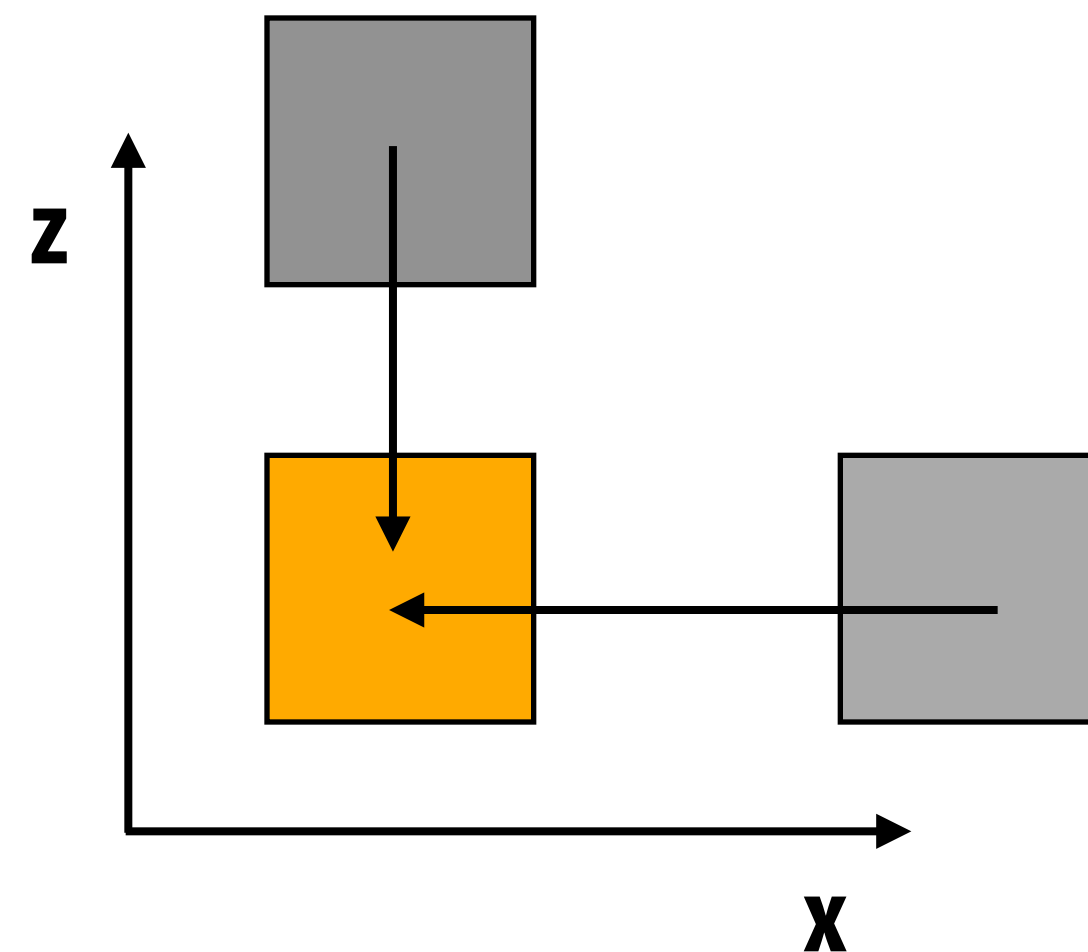
- Divide space into $b_x \times b_y \times b_z$ cells
- Want to avoid computing interaction between each particle pair twice
- Half-shell rules (applied in this order):
 - If $A_x < B_x$ interact in home box of A
 - If $A_y < B_y$ interact in home box of A
 - If $A_z < B_z$ interact in home box of A
 - If A and B are in the same box, no movement is necessary for interaction.

NT intuition



HS Method

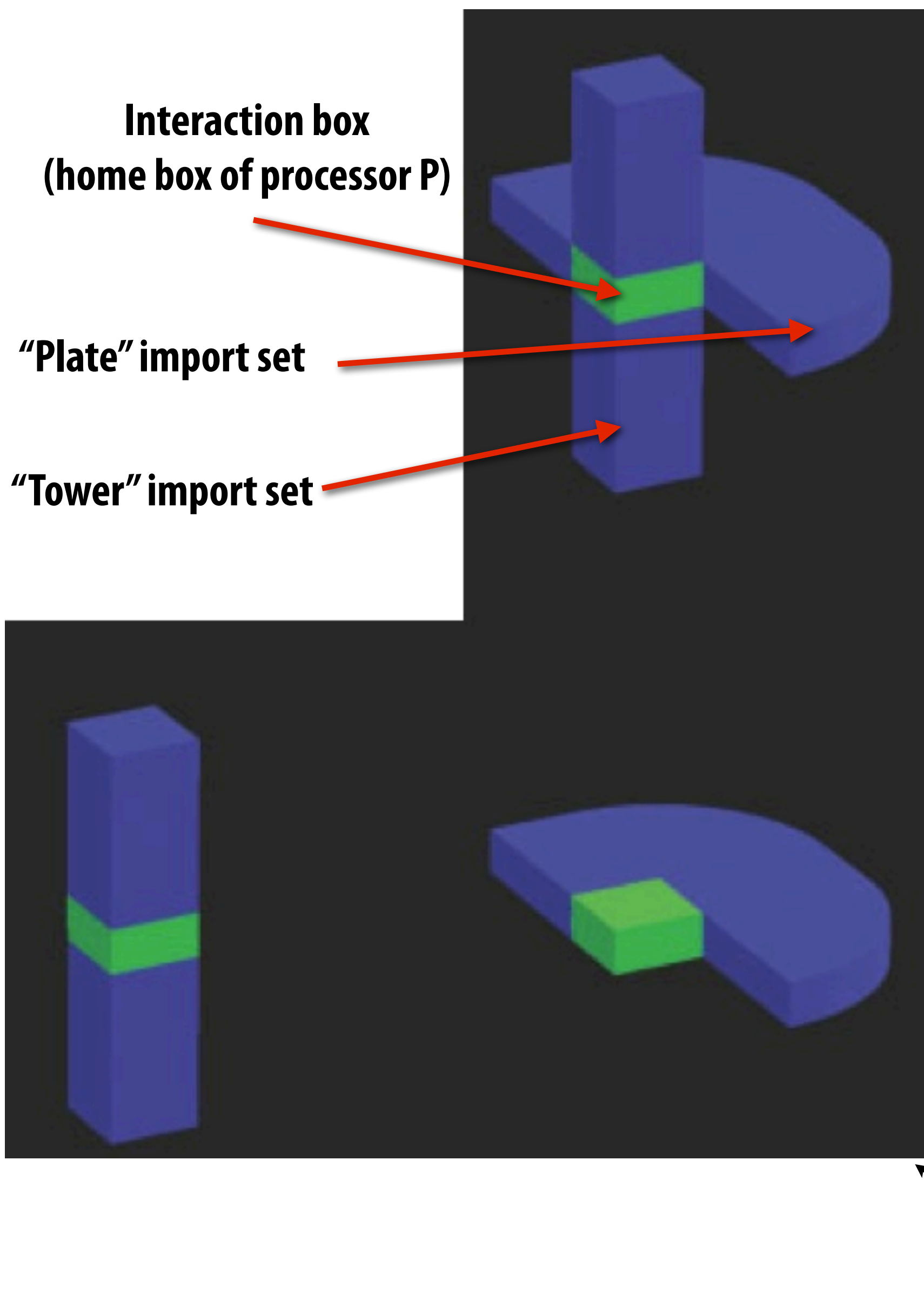
**Perform computation in box with
smaller X coordinate**



NT Method

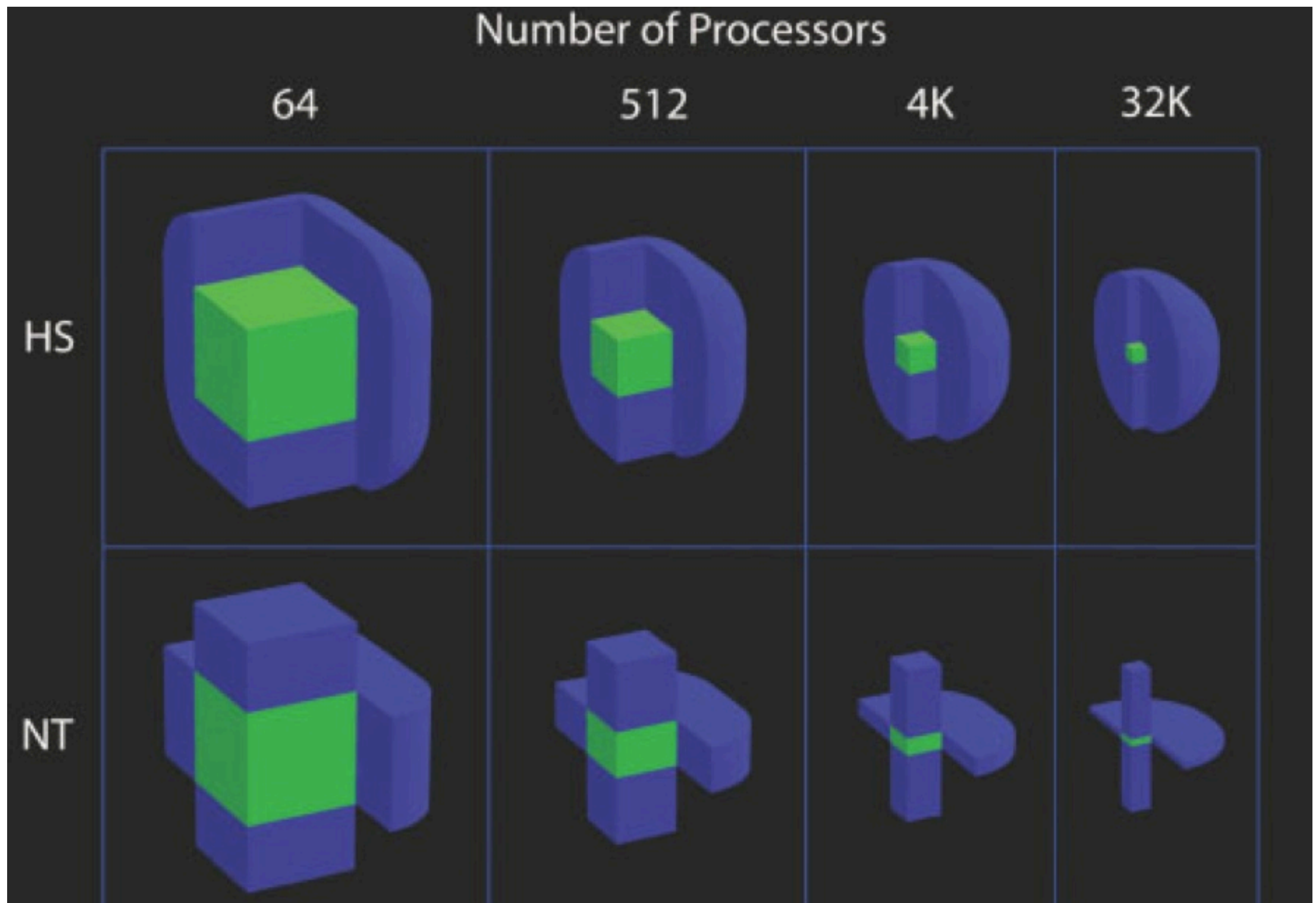
**Perform computation in box with X
coordinate from particle with smaller X
coordinate and Z coordinate from particle
with larger X coordinate**

Extending NT method to 3D



- Divide space into $b_x \times b_y \times b_z$ cells
- NT interaction rules (applied in this order):
 - If $A_x < B_x$ A is in the tower
 - If $A_y < B_y$ A is in the tower
 - If $A_z < B_z$ A is in the plate
 - If A and B are in the same box, choose arbitrarily
- Implication: two particles will interact in box with x,y coordinate of tower and z coordinate of plate)

Scaling



NT Method asymptotics

■ Half-shell method

- import volume $V = 3Rb^2 + (3/2)\pi R^2b + (2/3)\pi R^2$
- As $P \longrightarrow \infty$: $V = (2/3)\pi R^2$

■ NT-method

- import volume $V = 2Rb_{xy}^2 + 2Rb_z^2 + (1/2)\pi R^2b_z$
- As $P \longrightarrow \infty$: $V = O(R^{3/2} / p^{1/2})$

NT method summary

- **Communication-to-computation ratio increases as number of processors gets large**
- **Reduce communication requirements by ALWAYS communicating**
 - **pick a “neutral processor” to perform computation between two particles**