Data Parallelism (1)

Lecture 12

Hartmut Kaiser

https://teaching.hkaiser.org/spring2025/csc4700/

Today's Theme

- You are now accustomed to thinking about parallel programming in terms of "what workers do" and "assigning work to workers"
- Today I would like you to think about describing algorithms in terms of operations on sequences of data
  - `map`
  - `filter`
  - `fold / reduce`
  - `scan / segmented scan`
  - `sort`
  - `group_by`
  - `join`
  - `partition / flatten`
- Main idea: high-performance parallel implementations of these operations exist
  - So programs written in terms of these primitives can often run efficiently on a parallel machine (if you can avoid being bandwidth bound)

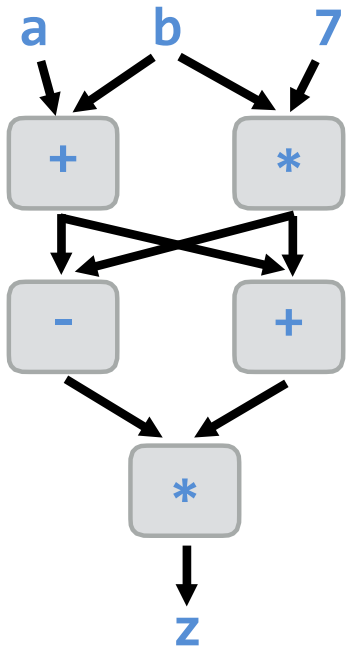
Motivation

- Why must an application expose large amounts of parallelism?
  - Utilize large numbers of cores
  - High-core count machines
  - Many machines (e.g., cluster of machines in the cloud)
  - SIMD processing + multi-threaded cores require even more parallelism
  - GPU architectures require very large amounts of parallelism

Understanding Dependencies is Key

- Key part of parallel programming is understanding when dependencies exist between operation
- Lack of dependencies implies potential for parallel execution

```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



Data-Parallel Model

- Organize computation as operations on sequences of elements
  - e.g., perform same function on all elements of a sequence
  - Helps decoupling steps in computation
- Data parallelism is parallelization across multiple processors in parallel computing environments
  - It focuses on distributing the data across different computational units, which operate on the data in parallel
- A well-known modern example:
  - Numpy: `C = A + B` (A, B, and C are vectors of same length)
- We will look at various algorithms that work on ranges of elements
- Many common problems can be solved by applying some of these algorithms

Data-Parallel Model

Data parallelism	Task parallelism
Same operations are performed on different subsets of same data	Different operations are performed on the same or different data
Synchronous computation	Asynchronous computation
Speedup is more as there is only one type of execution thread operating on all sets of data	Speedup is less as each processor will execute a different thread or process on the same or different set of data
Amount of parallelization is proportional to the input data size	Amount of parallelization is proportional to the number of independent tasks to be performed
Designed for optimum load balance on multi processor system	Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling

Key Data Type: Sequences

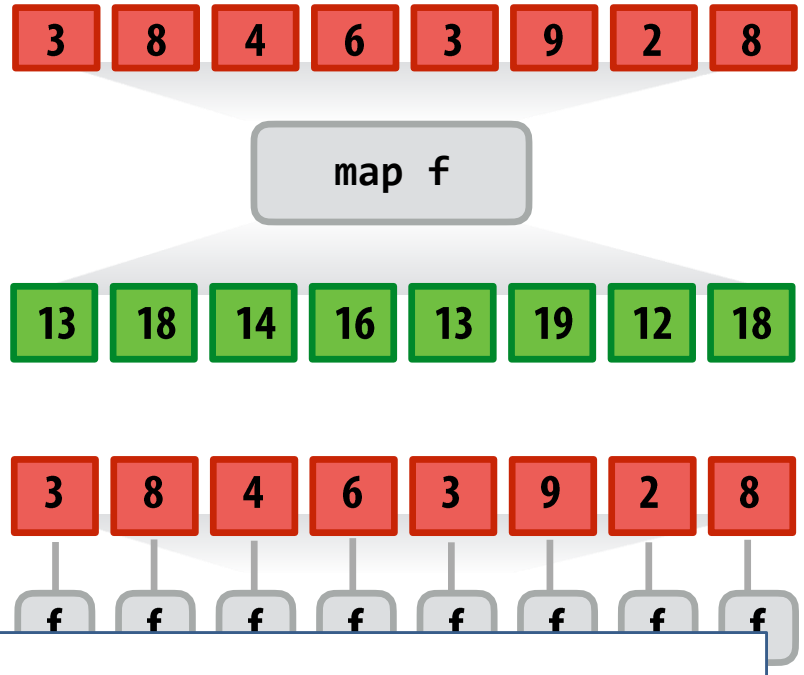
- Collection of elements
  - In C++: `vector<T>` (actually any C++ container)
  - Similar constructs exist in other programming languages:
    - Scala lists: `List[T]`
    - Python: Pandas Dataframes, Numpy arrays
    - In a functional language (like Haskell): `seq T`
    - Etc.

- Important: data-parallel programs only access elements of a sequence through specific operations, not direct element access
  - C++: Iterators

Map

- Higher order function (function that takes a function as an argument)
- Applies side-effect free unary function `f :: a -> b` to all elements of input sequence, producing an output sequence of the same length
- In a functional language (e.g., Haskell):
  - `map :: (a -> b) -> seq a -> seq b`
- In C++:

```
template <typename InputIt, typename Ou
          typename UnaryOp>
OutputIt transform(
    InputIt first1, InputIt last1,
    OutputIt d_first, UnaryOp unary_op)
```



C++

```
int f(int x) { return x + 10; }

int a[] = {3, 8, 4, 6, 3, 9, 2, 8};
int b[8];
std::transform(a, a+8, b, f);

Produces:
{13, 18, 14, 16, 13, 19, 12, 18};
```

Output start iterator

Input end iterator

Input start iterator

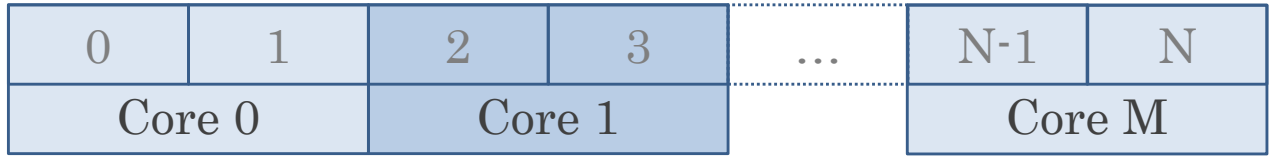
Parallelizing Map

- Since `f :: a -> b` is a side-effect free function, applying `f` to all elements of the sequence can be performed
  - In any order without changing the output of the program
  - Therefore, the implementation of map has the flexibility to reorder/parallelize processing of elements of the sequence however it sees fit
  - Side-effect free → no multi-threading issues!

```
map f s =
    partition sequence s into P smaller subsequences s_i
    for each subsequence s_i (in parallel)
        out_i = map f s_i
    out = concatenate out_i's
```

Parallelizing Map

Sequence of elements:

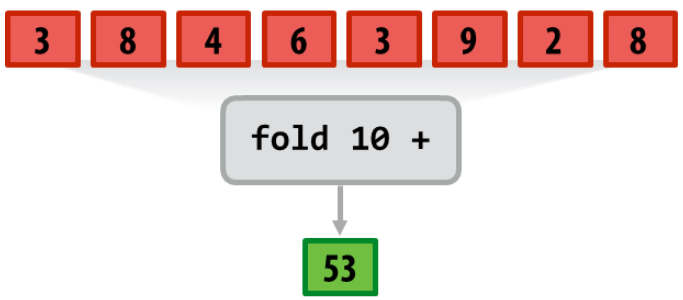


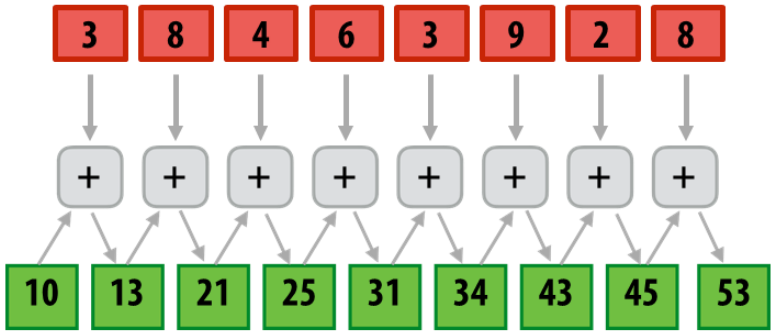

```
std::vector<int> d = {...}, target(size);
std::transform(std::execution::par, d.begin(), d.end(), target.begin(), [](int val) {...});
```

```
template <typename InIter, typename OutIter, typename F>
void transform(std::execution::parallel_policy, InIter b, InIter e, OutIter dest, F f)
{
    auto size = std::distance(b, e);                          // Iterator should be random access
    std::vector<std::jthread> v;
    for (size_t chunk = 0; chunk != NUM_CHUNKS; ++chunk) {    // assume: cleanly divisible
        v.push_back(std::jthread([=]() {                      // launch new thread for each chunk
            auto begin = std::next(b, (chunk * size) / NUM_CHUNKS);
            std::transform(begin, std::next(begin, size / NUM_CHUNKS),
                std::next(dest, (chunk * size) / NUM_CHUNKS), f);   // sequential transform()
        }));
    }
}     // join threads implicitly
```

Fold (fold left)

- Apply binary operation op to each element and an accumulated value
  - Seeded by initial value `T init`



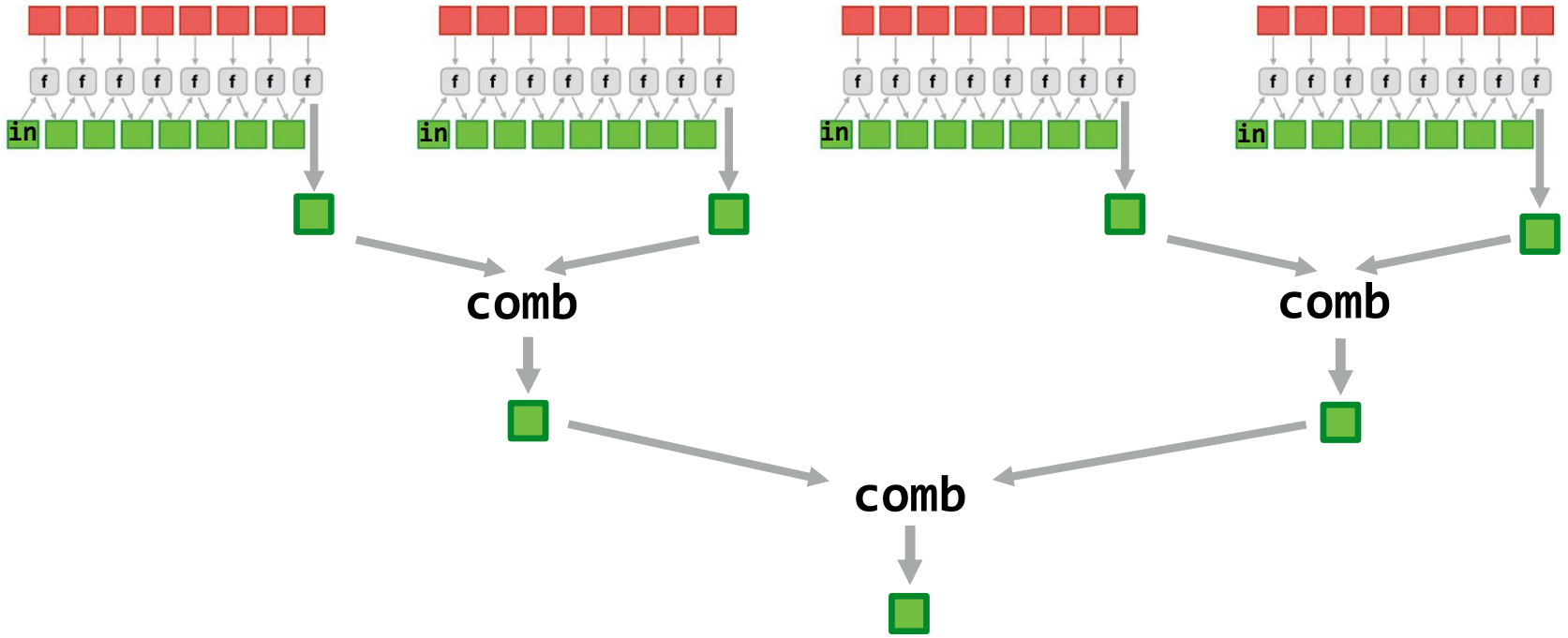



```
template <typename InputIt, typename T, typename BinaryOp>
T reduce(InputIt first, InputIt last, T init, BinaryOp op);
```

```
std::vector<int> numbers = {3, 8, 4, 6, 3, 9, 2, 8};
int sum = std::ranges::reduce(numbers, 10);
std::println("Sum: {}", sum);     // Outputs: Sum: 53
```

Parallel Fold (fold left)

- Apply `f` to each element and an accumulated value
- In addition to binary function `f`, also need an additional binary "combiner" function (no need for `comb` if `f` is an associative function)
- Seeded by initial value of type `b` (must be identity for `f` and `comb`)



Parallel Fold (fold left)

- In C++:

```
template <typename ExecutionPolicy, typename ForwardIt, typename T>
T reduce(ExecutionPolicy&& policy,
    ForwardIt first, ForwardIt last, T init);      // uses std::plus<T>

template <typename ExecutionPolicy, typename ForwardIt, typename T,
    typename BinaryOp>
T reduce(ExecutionPolicy&& policy,
    ForwardIt first, ForwardIt last, T init, BinaryOp op);
```

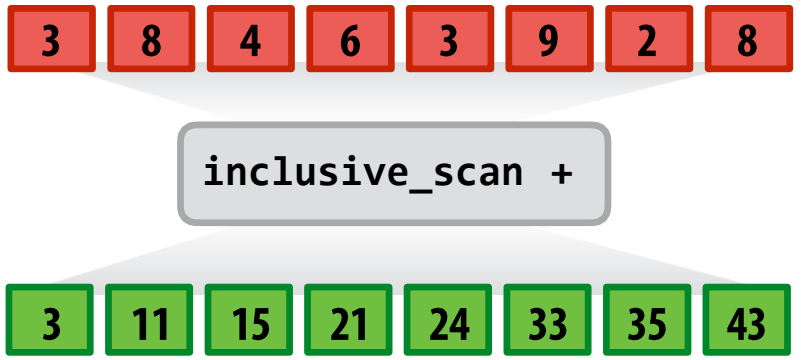
```
std::vector<int> numbers = {3, 8, 4, 6, 3, 9, 2, 8};
int sum = std::reduce(std::execution::par, numbers.begin(), numbers.end(), 10);
std::println("Sum: {}", sum);    // Outputs: Sum: 53
```

Scan

- Given input, produce output where:
  - $output[i] = input[0] \oplus input[1] \oplus \cdots \oplus input[i]$
  - Where $\oplus$: associative operator (not necessarily commutative)
- Running totals: $y_0 = x_0, y_1 = y_0 + x_1, y_2 = y_1 + x_2, \ldots$
- Example (using operator +):
  - `Input:  [6  4 16 10 16 14  2  8]`
  - `Output: [6 10 26 36 52 66 68 76]`
- Sequential scan:

```
inclusive_scan(float in[], float out[], int N) {
    out[0] = in[0];
    for (i = 1; i < N; ++i) out[i] = out[i-1] + in[i];
}
```

- Does not seem parallelizable
  - Work: $O(n)$ - total operations performed
  - Span: $O(n)$ - longest chain of sequential steps
- This algorithm is sequential, but a different algorithm has Work: $O(n)$, Span: $O(\log n)$



Inclusive Scan

```
float op(float a, float b) { … }
inclusive_scan(float in[], float out[], int N) {
    out[0] = in[0];
    for (i = 1; i < N; ++i)
        out[i] = op(out[i-1], in[i]);
}
```

- In C++:

```
template <typename InputIt, typename OutputIt, typename BinaryOp>
OutputIt inclusive_scan(InputIt first, InputIt last,
    OutputIt d_first, BinaryOp op);    // op defaults to std::plus<>

template <typename InputIt, typename OutputIt, typename BinaryOp, typename T>
OutputIt inclusive_scan(InputIt first, InputIt last,
    OutputIt d_first, BinaryOp op, T init);
```

Exclusive Scan

- Alternative form: "exclusive scan": `out[i]` is the scan result for all elements up to, but excluding, `in[i]`:
  - Running totals: $y_0 = 0$, $y_1 = y_0 + x_0$, $y_2 = y_1 + x_1$, …

```
float op(float a, float b) { … }
exclusive_scan(float in[], float out[], int N, float init) {
    out[0] = init;
    for (i = 1; i < N; ++i)
        out[i] = op(out[i-1], in[i-1]);
}
```

- In C++:

```
template <typename InputIt, typename OutputIt, typename T, typename BinaryOp>
OutputIt exclusive_scan (InputIt first, InputIt last,
    OutputIt d_first, T init, BinaryOp op);
```

Scan

- Scan is a pattern that arises in many, many problems
- Examples:
  - Running minimum or maximum of all elements to the left of i^{th}
  - Is there an element to the left of i^{th} element satisfying some property?
  - Count of elements to the left of i^{th} element satisfying some property
    - This last one is perfect for an efficient parallel pack (filter)…
    - Perfect for building on top of the "parallel prefix trick"
  - Generating row-indices for CSR/CSC matrix format
  - Find the number of subarrays with a specific sum
    - … and many other interview questions can be handled by applying a (variation) of a scan algorithm

Parallel Scan

Data-Parallel Scan

let $A = [a_0, a_1, a_2, \ldots, an]$

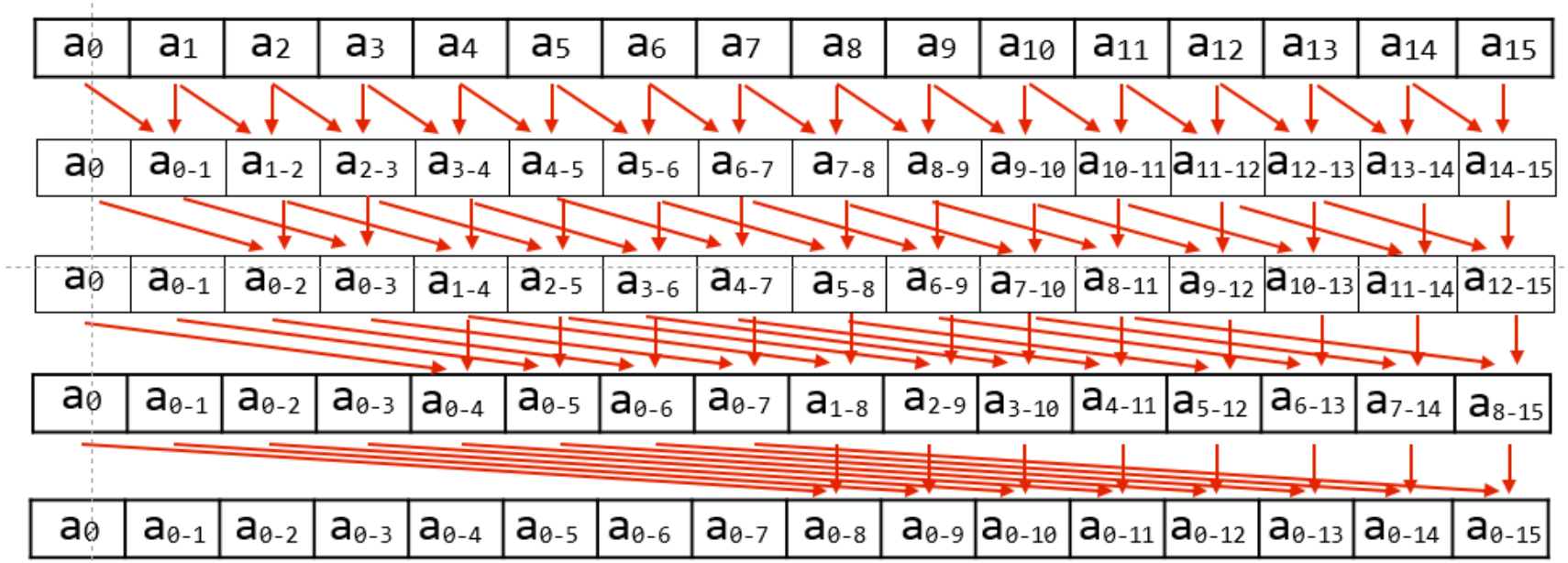
let $\oplus$ be an associative binary operator with identity element I

$$inclusive\_scan(\oplus, A) = [a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \ldots]$$
$$exclusive\_scan(\oplus, A) = [I, a_0, a_0 \oplus a_1, \ldots]$$

- If operator is +, then $inclusive\_scan(+, A)$ is called "a prefix sum":

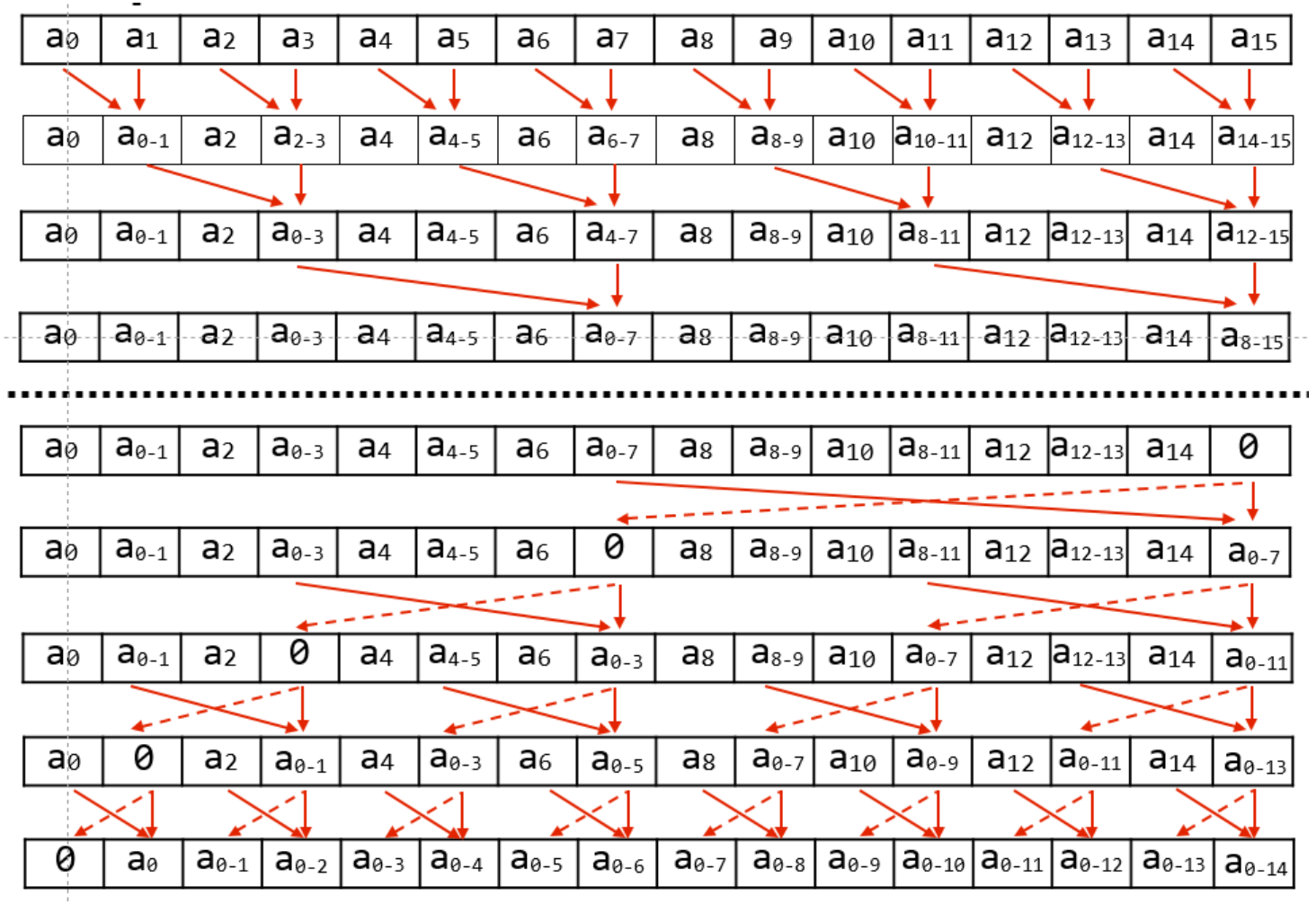
$$prefix\_sum(A) = [a_0, a_0 + a_1, a_0 + a_1 + a_2, \ldots]$$

Data-Parallel Inclusive Scan



- Inefficient compared to sequential algorithm:
  - Total operations performed: $O(N \log N)$
  - Longest chain of sequential steps: $O(\log N)$

Work-efficient Parallel Exclusive Scan ($O(N)$ work, $O(LogN)$ span)



Work efficient exclusive Scan Algorithm

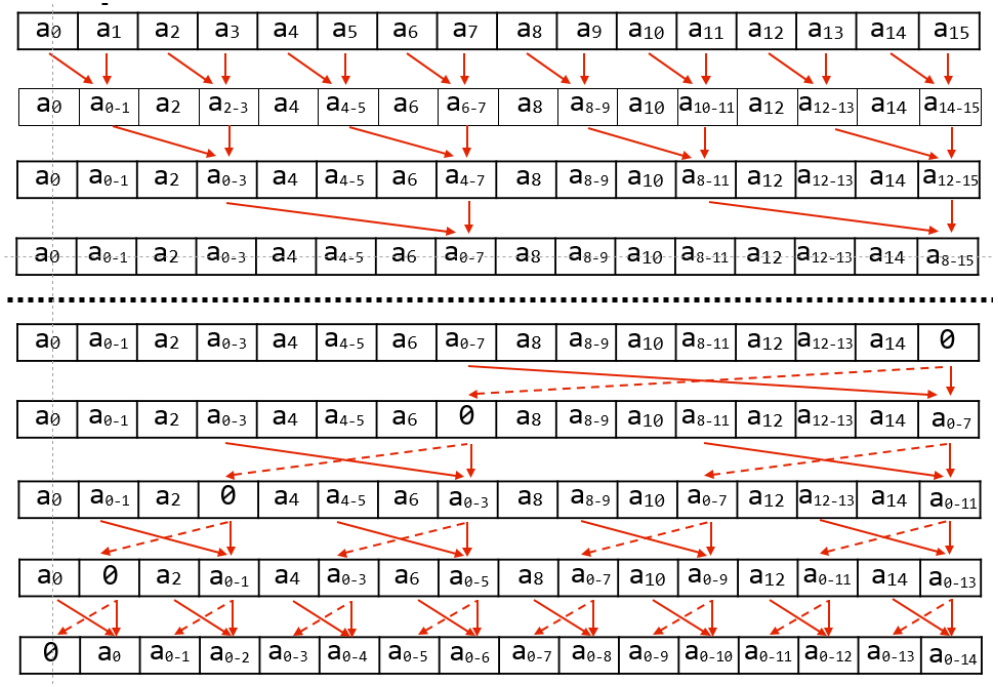
Up-sweep:

```
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:

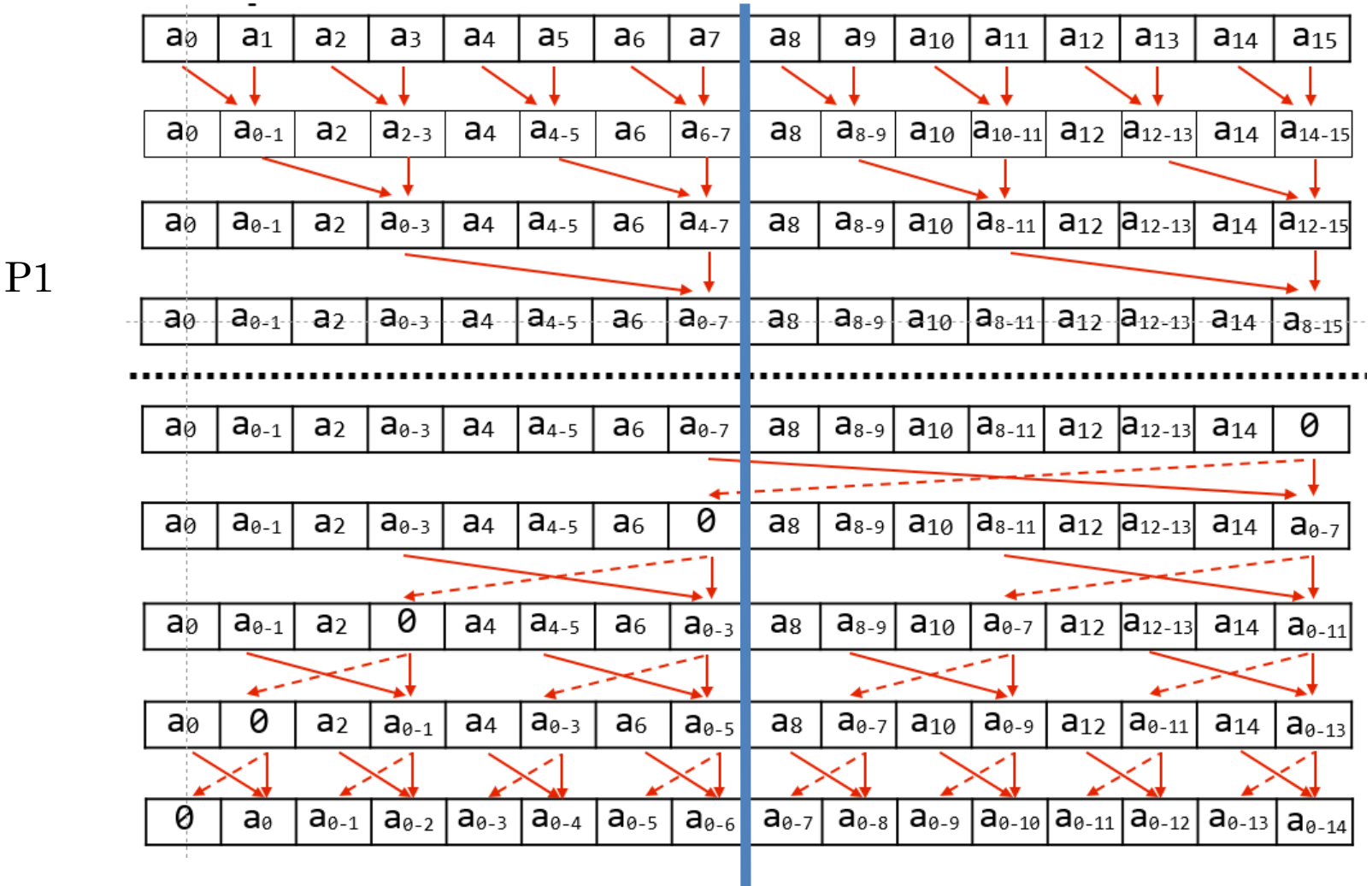
```
x[n-1] = 0
for d=(log2n - 1) down to 0 do
   forall k=0 to n-1 by 2^(d+1)    do
      tmp = a[k + 2^d - 1]
      a[k + 2^d - 1] = a[k + 2^(d+1) - 1]
      a[k + 2^(d+1) - 1] = tmp + a[k + 2^(d+1) - 1]
```

(with ⊕="+")



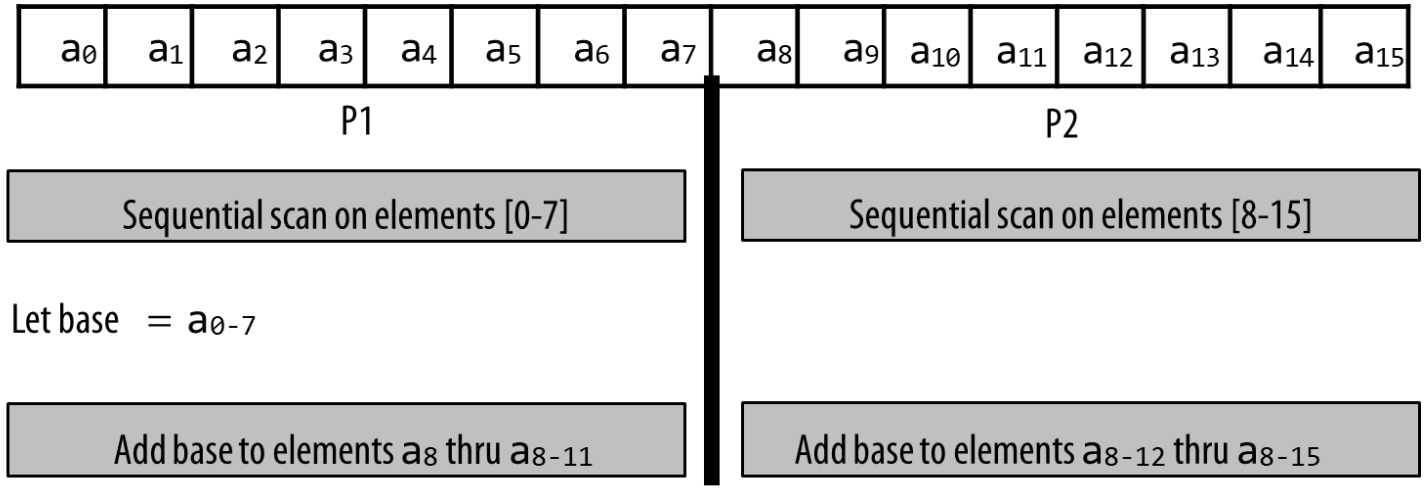
Work: $O(N)$
Span: $O(\log N)$
Locality: ??

Now consider Scan Implementation on just two Cores



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 $a_{8\text{-}11}$ may be more costly on large core count system with non-uniform memory access costs, but on small-scale
  - On multi-core system the access cost is likely almost the same as from P2



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 in algorithm
  - Different strategy for performing scan at different levels of the machine
  - 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 a sequence of sequences
- Examples:
  - For each vertex `v` in a graph
    - For each edge `e` connected to `v`
  - For each particle `p` in a simulation
    - For each particle within distance `D` of `p`
  - For each document `d` in a collection
    - For each word in `d`
- There are two levels of parallelism in the problem that a programmer might want to exploit
- But it is irregular: the size of edge lists, particle neighbor lists, words per document, etc,
  - May be very different from vertex to vertex (or particle to particle)
  - Parallelization over subsequences may not scale well

Segmented Scan

- Generalization of scan
- Simultaneously perform scans on contiguous partitions of input sequence

  let $A = [[1,2],[6],[1,2,3,4]]$

  let $\oplus$ be +

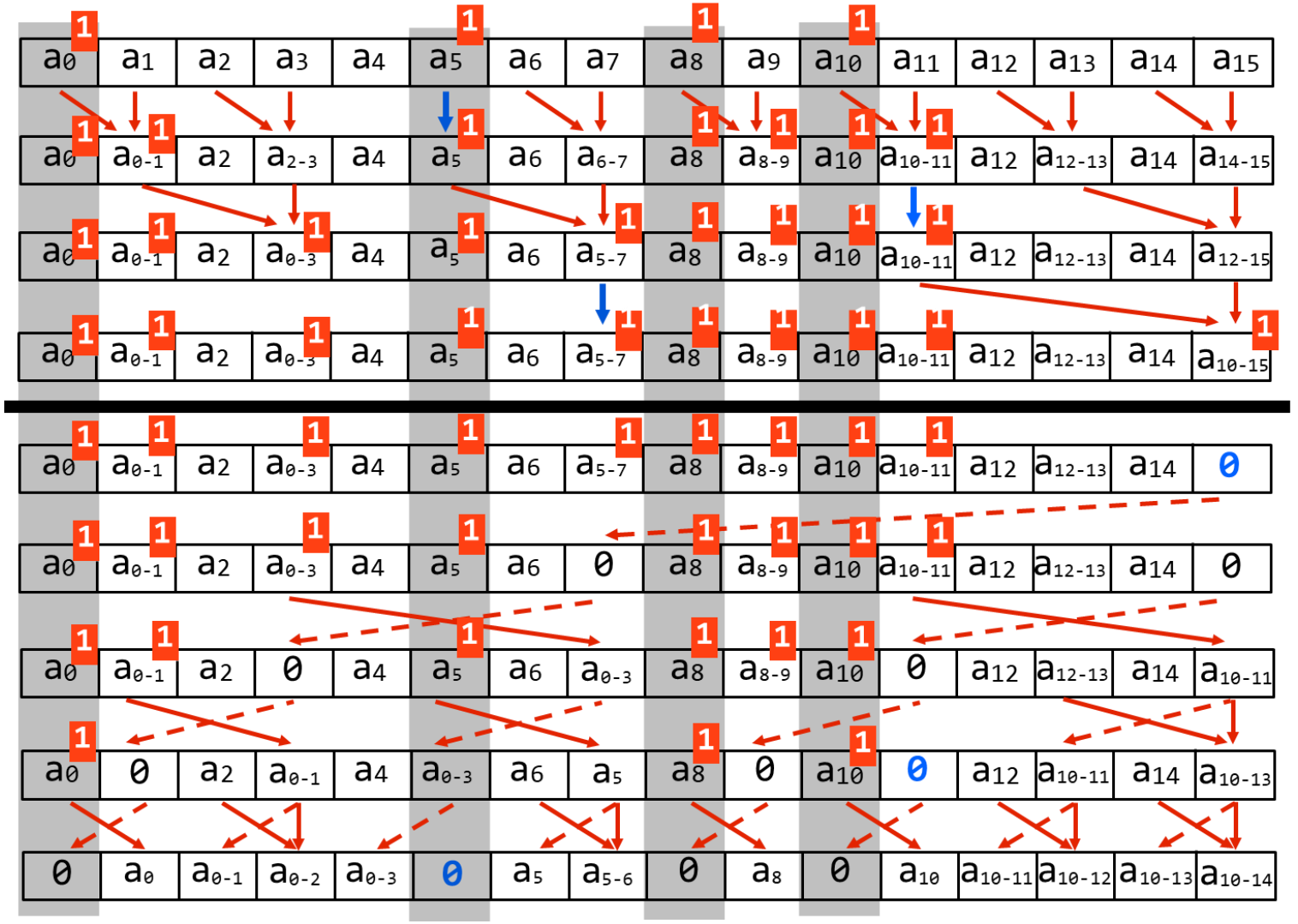
  $segmented\_exclusive\_scan(\oplus, A) = [[0,1],[0],[0,1,3,6]]$

- Assume a simple "start-flag" representation of nested sequences:
  - Consider nested sequence: $A = [[1,2,3],[4,5,6,7,8]]$
  - Flag: 1 0 0 1 0 0 0 0
  - Data: 1 2 3 4 5 6 7 8

Segmented Scan

- Instead of applying $\oplus$ to data A:
  - $exclusive\_scan(\oplus, A) = [I, a_0, a_0 \oplus a_1, \dots]$
- Apply $\oplus$ to pairs of input (data A and flag F):
  - $exclusive\_scan(\oplus, \left\langle \begin{matrix} A \\ F \end{matrix} \right\rangle) = [I, \left\langle \begin{matrix} a_0 \\ f_0 \end{matrix} \right\rangle, \left\langle \begin{matrix} a_0 \\ f_0 \end{matrix} \right\rangle \oplus \left\langle \begin{matrix} a_1 \\ f_1 \end{matrix} \right\rangle, \dots]$
  - Where: $\left\langle \begin{matrix} a_i \\ f_i \end{matrix} \right\rangle \oplus \left\langle \begin{matrix} a_k \\ f_k \end{matrix} \right\rangle = \left\langle \begin{matrix} a_i \times \overline{f_i} \oplus a_k \\ f_i \vee f_k \end{matrix} \right\rangle$ $\overline{f_i}$: 'not' f_i
  - If flag is zero: inner element, i.e. do normal scan operation
  - If flag is non-zero: start a new scan operation
- Then the upper row of result will represent segmented scan result

Segmented Scan (exclusive)



Work-efficient segmented Scan

Up-sweep: (with ⊕="+")

```
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
```

Sparse Matrix Multiplication

- Most values in matrix are zero
  - Note: easy parallelization by parallelizing the different per-row dot products
  - But different amounts of work per row (complicates wide SIMD execution)
- Example sparse storage format: compressed sparse row
  - $values = [\,[3,1],[2],[4],\ldots,[2,6,8]\,]$
  - $cols = [\,[0,2],[1],[2],\ldots.,]$
  - $row\_starts = [0,2,3,4,\ldots\,]$

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

Sparse Matrix Multiplication with Scan

- Example:
  - $x = [x_0, x_1, x_2, x_3]$
  - $values = [\,[3,1],[2],[4],[2,6,8]\,]$
  - $cols = [\,[0,2],[1],[2],[1,2,3]\,]$
  - $row\_starts = [0,2,3,4]$

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

- Map over all non-zero values: $products[i] = values[i] * x[cols[i]]$
  - products = $[3x_0, x_2, 2x_1, 4x_2, 2x_1, 6x_2, 8x_3]$

gather(x, cols)

- Create flags vector from row_starts: $flags = [1,0,1,1,1,0,0]$
- Perform inclusive segmented-scan on (products, flags) using addition operator
  - $[3x_0, 3x_0 + x_2, 2x_1, 4x_2, 2x_1, 2x_1 + 6x_2, 2x_1 + 6x_2 + 8x_3]$
- Take last element in each segment:
  - $y = [3x_0 + x_2, 2x_1, 4x_2, 2x_1 + 6x_2 + 8x_3]$

LSU CENTER FOR COMPUTATION & TECHNOLOGY

STE||AR GROUP