AMS 250: An Introduction to High Performance Computing

Map and Reduce Patterns

Shawfeng Dong

shaw@ucsc.edu

(831) 459-2725

Astronomy & Astrophysics

University of California, Santa Cruz

Outline

- Map pattern
  - Optimizations
  - Example: Scaled Vector Addition (SAXPY)
- Collectives
  - Reduce Pattern
  - Scan Pattern
  - Gather Pattern
  - Scatter Pattern
  - Pack Pattern

Mapping

- “Do the same thing many times”

```
foreach i in foo:
    do something
```

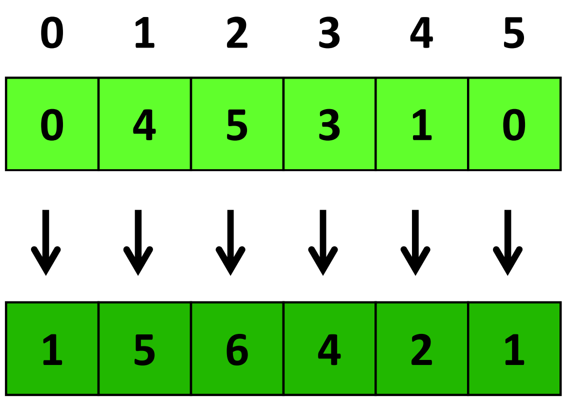
- Well-known higher order function in Functional Languages like Haskell, ML, Scala:

```
map :: (a -> b) -> [a] -> [b]
```

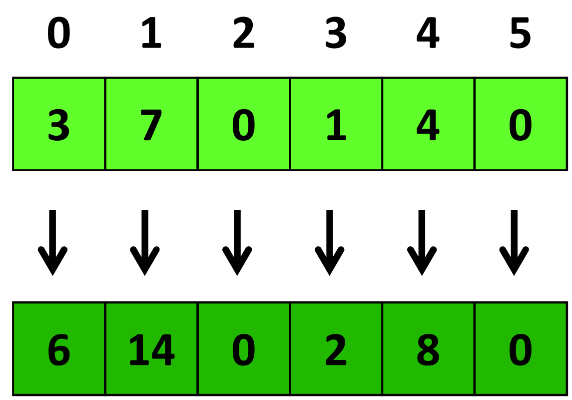
applies a function to each element in a list and returns a list of results

Example Maps

Add 1 to every item in an array



Double every item in an array



Key Point: An operation is a map if it can be applied to each element without knowledge of neighbors.

Key Idea

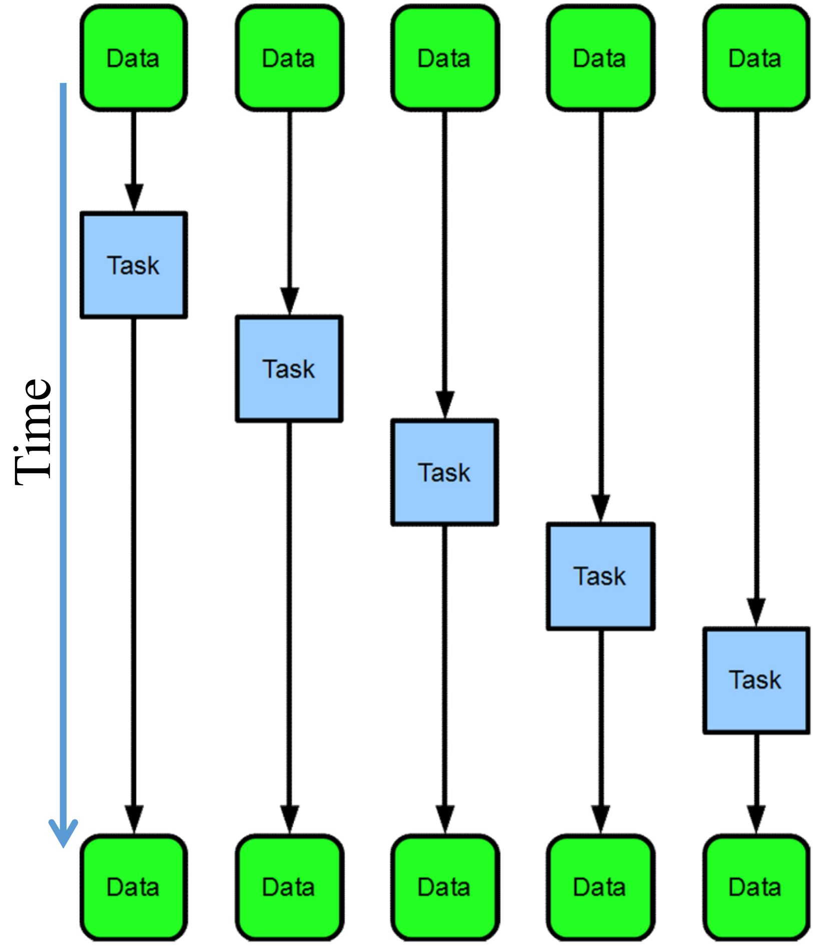
- Map is a "foreach loop" where each iteration is independent

Embarrassingly Parallel

Independence is a big win. We can run map completely in parallel. Significant speedups! More precisely: $T(n)$ is $O(1)$ plus implementation overhead that is $O(\log n)$; so $T(n) = O(\log n)$.

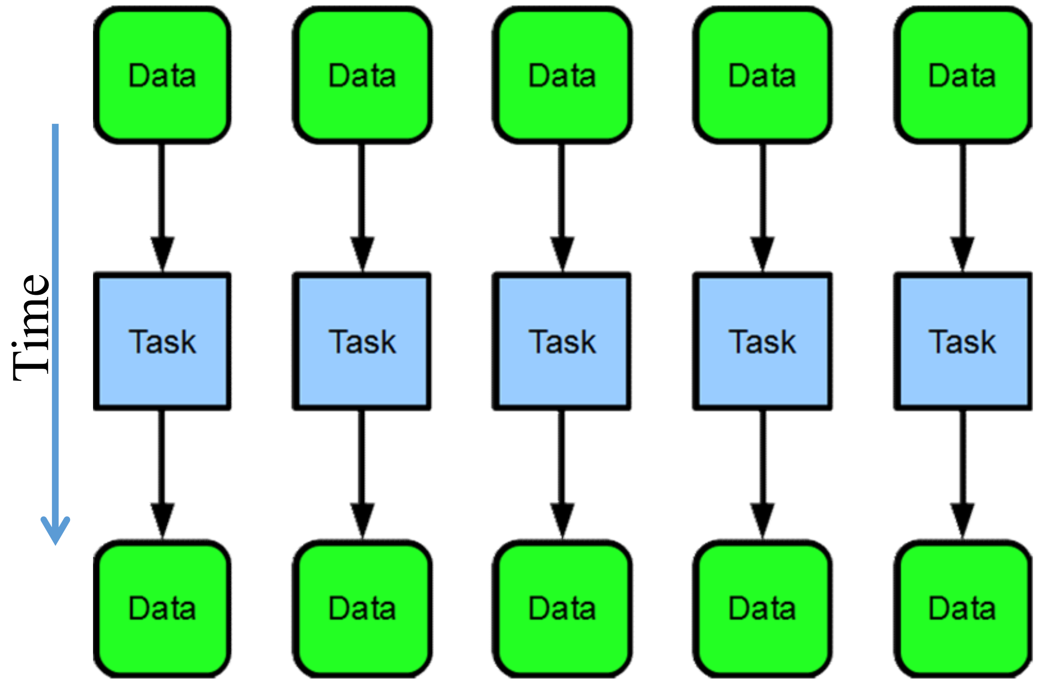
Sequential Map

```
for (int n=0; n< array.length; ++n)
{
    process(array[n]);
}
```



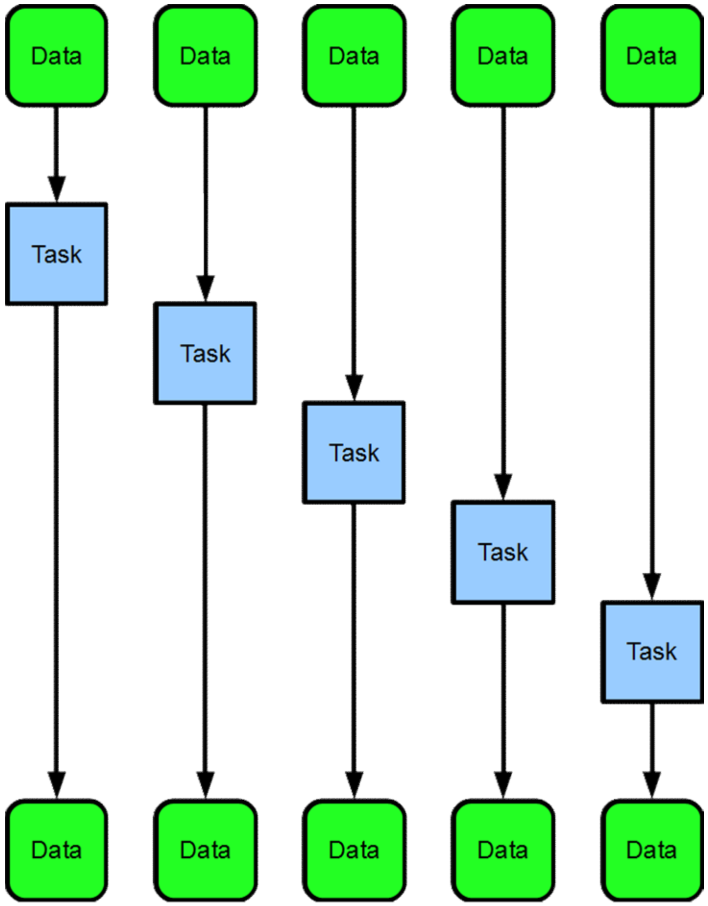
Parallel Map

```
parallel_for_each (x in array)
{
    process(x);
}
```

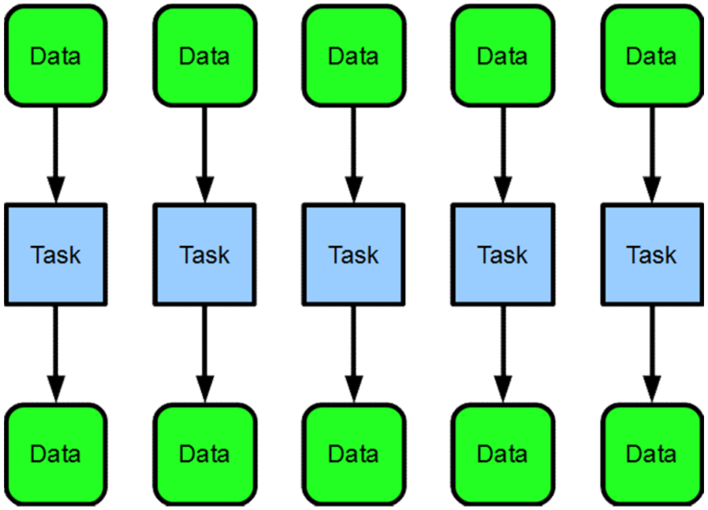


Comparing Maps

Serial Map



Parallel Map



Speedup

The space here is speedup. With the parallel map, our program finished execution early, while the serial map is still running.

Independence

- The key to (embarrassing) parallelism is independence

Warning: No shared state!

Map function should be “pure” (or “pure-ish”) and should not modify shared states

- Modifying shared state breaks perfect independence
- Results of accidentally violating independence:
  - non-determinism
  - data-races
  - undefined behavior
  - segfaults

Implementation and API

- OpenMP and Cilk Plus contain a parallel ***for*** language construct
- Map is a mode of use of parallel ***for***
- TBB uses **higher order functions** with lambda expressions/"functors"
- Some languages (Cilk Plus, Matlab, Fortran) provide **array notation** which makes some maps more concise

Array Notation

```
A[:] = A[:]*5;
```

is Cilk Plus array notation for "multiply every element in *A* by 5"

Unary Maps

Unary Maps

So far we have only dealt with mapping over a single collection…

Map with 1 Input and 1 Output

	0	1	2	3	4	5	6	7	8	9	10	11
x	3	7	0	1	4	0	0	4	5	3	1	0
result	6	14	0	2	8	0	0	8	10	6	2	0

```
int oneToOne ( int x[11] ) {
    return x*2;
}
```

N-ary Maps

N-ary Maps

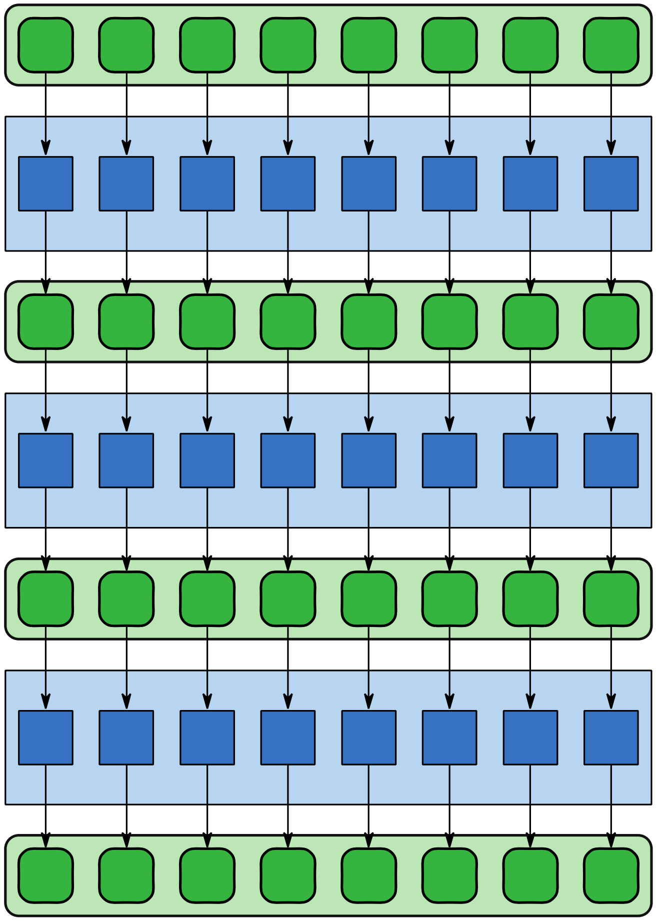
But sometimes it makes sense to map over multiple collections at once…

Map with 2 Inputs and 1 Output

	0	1	2	3	4	5	6	7	8	9	10	11
x	3	7	0	1	4	0	0	4	5	3	1	0
y	2	4	2	1	8	3	9	5	5	1	2	1
result	5	11	2	2	12	3	9	9	10	4	3	1

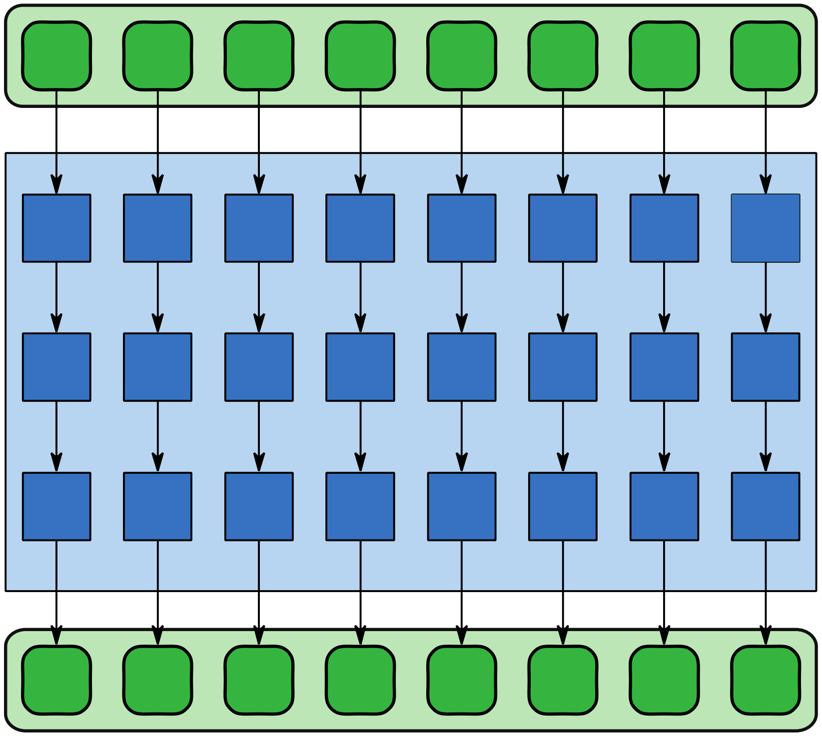
```
int twoToOne ( int x[11], int y[11] ) {
    return x+y;
}
```

Optimization – Sequences of Maps



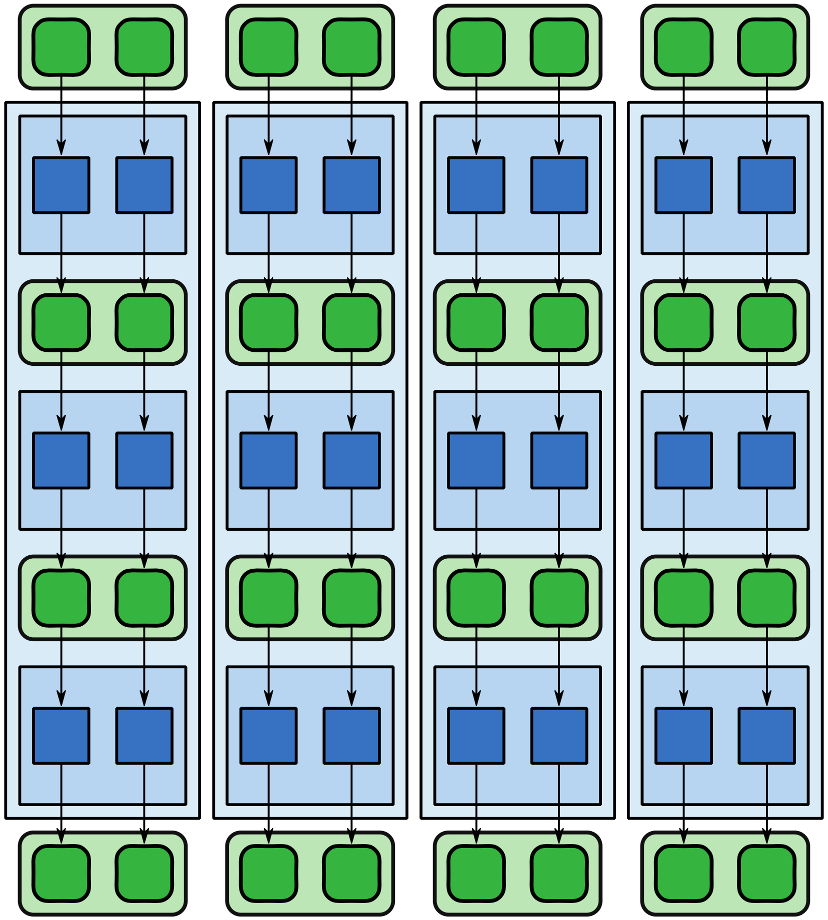
- Often several map operations occur in sequence
  - Vector math consists of many small operations such as additions and multiplications applied as maps
- A naïve implementation may write each intermediate result to memory, wasting memory bandwidth and likely overwhelming the cache

Optimization – Code Fusion



- Can sometimes “fuse” together the operations to perform them at once
- Adds arithmetic intensity, reduces memory/cache usage
- Ideally, operations can be performed using registers alone

Optimization – Cache Fusion



- Sometimes impractical to fuse together the map operations
- Can instead break the work into blocks, giving each CPU one block at a time
- Hopefully, operations use cache alone

Example: Scaled Vector Addition (SAXPY)

- y ← ax + y
  - Scales vector x by a and adds it to vector y
  - Result is stored in input vector y
- A level-1 routine in the **BLAS** (Basic Linear Algebra Subprograms) library
- **Every element in vector x and vector y are independent**

What does $y \leftarrow ax + y$ look like?

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

$y \leftarrow ax + y$ Visual:

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Twelve processors used → one for each element in the vector

$y \leftarrow ax + y$ Visual:

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Six processors used → one for every two elements in the vector

$y \leftarrow ax + y$ Visual:

	0	1	2	3	4	5	6	7	8	9	10	11
a	4	4	4	4	4	4	4	4	4	4	4	4
*												
x	2	4	2	1	8	3	9	5	5	1	2	1
+												
y	3	7	0	1	4	0	0	4	5	3	1	0
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
y	11	23	8	5	36	12	36	49	50	7	9	4

Two processors used → one for every six elements in the vector

Serial SAXPY Implementation

```
void saxpy_serial(
    size_t n,              // the number of elements in the vectors
    float a,               // scale factor
    const float x[],       // the first input vector
    float y[]              // the output vector and second input vector
) {
    for (size_t i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

OpenMP SAXPY Implentation

```
void saxpy_openmp(
    int n,        // the number of elements in the vectors
    float a,      // scale factor
    float x[],    // the first input vector
    float y[]     // the output vector and second input vector
) {
#pragma omp parallel for
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

TBB SAXPY Implementation

```
void saxpy_tbb(
    int n,        // the number of elements in the vectors
    float a,      // scale factor
    float x[],    // the first input vector
    float y[]     // the output vector and second input vector
) {
    tbb::parallel_for(
        tbb::blocked_range<int>(0, n),
        [&](tbb::blocked_range<int> r) {
            for (size_t i = r.begin(); i != r.end(); ++i)
                y[i] = a * x[i] + y[i];
        }
    );
}
```

Cilk Plus SAXPY Implementation

```
void saxpy_cilk(
    int n,        // the number of elements in the vectors
    float a,      // scale factor
    float x[],    // the first input vector
    float y[]     // the output vector and second input vector
) {
    cilk_for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

Collectives

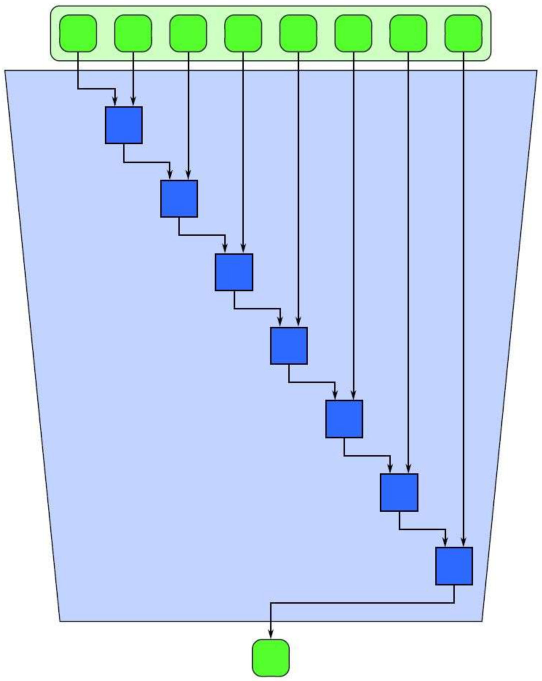
- **Collective** operations deal with a *collection* of data as a whole, rather than as separate elements
- Collective patterns include:
  - **Reduce**
  - **Scan**
  - **Gather**
  - **Scatter**
  - **Pack**

Reduce

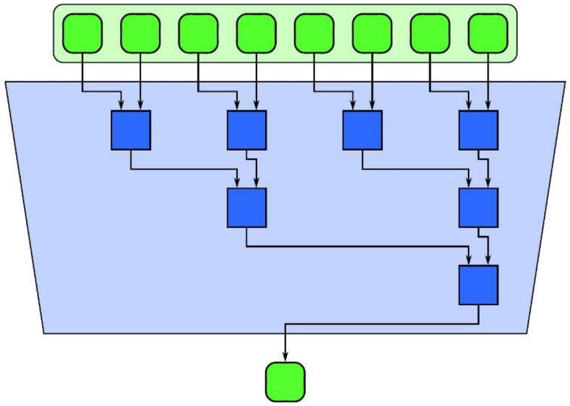
- **Reduce** is used to combine a collection of elements into one summary value
- A combiner function combines elements pairwise
- A combiner function only needs to be *associative* to be parallelizable
- Example combiner functions:
  - Addition
  - Multiplication
  - Maximum / Minimum

Serial vs. Parallel Reduce

Serial Reduction

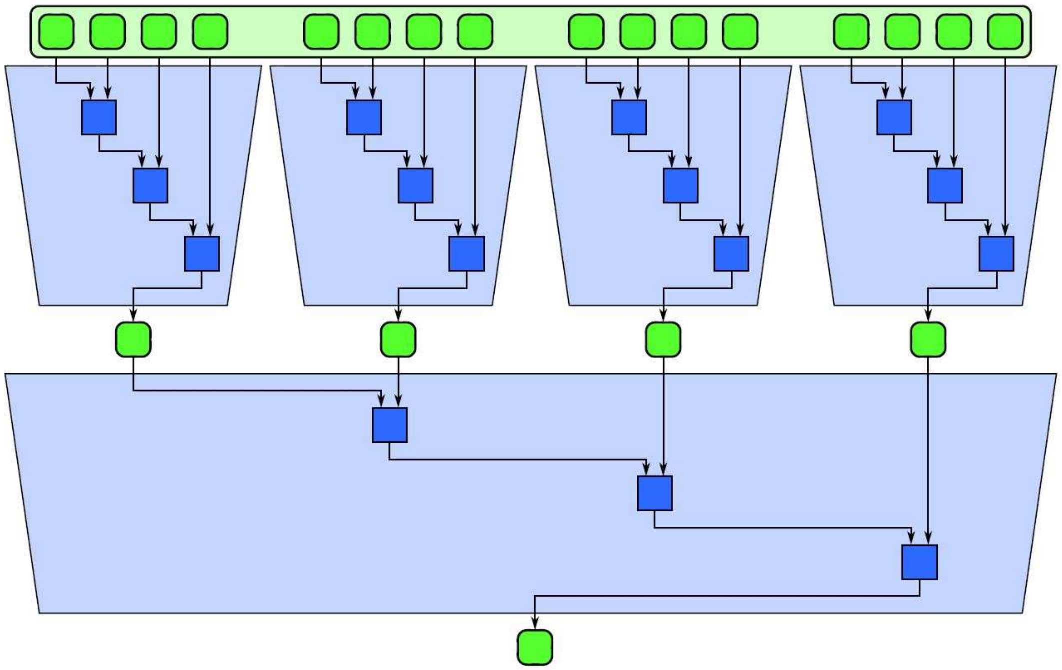


Parallel Reduction

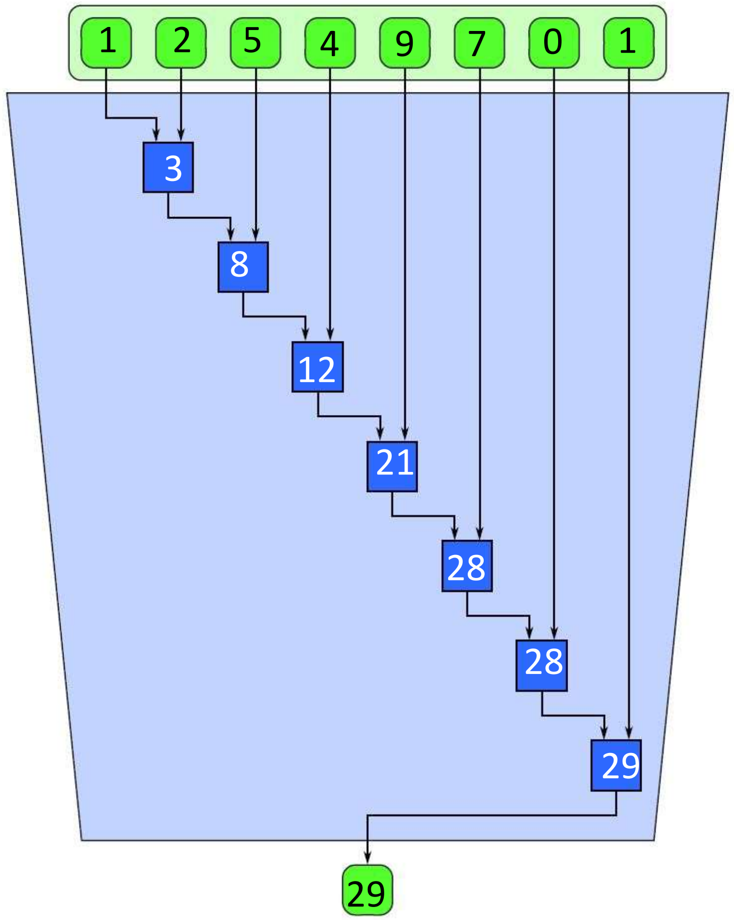


Tiled Reduce

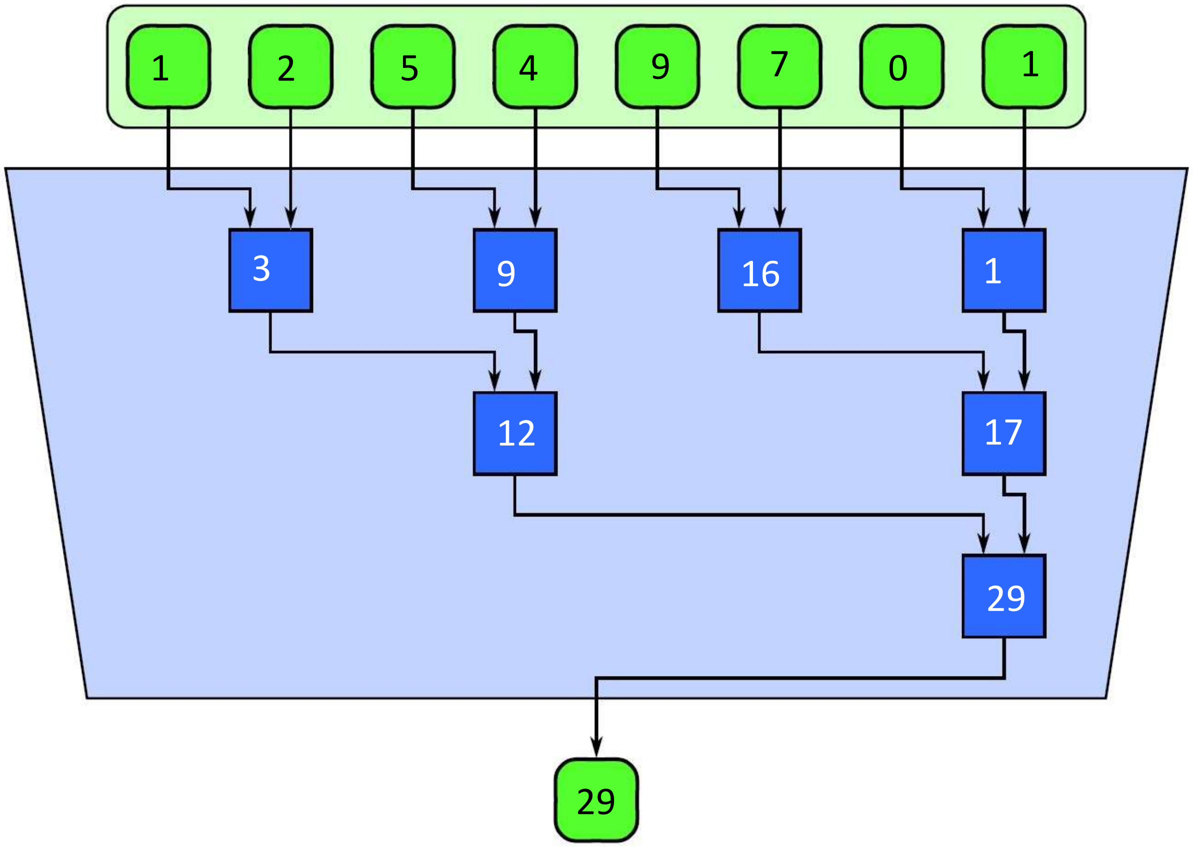
- **Tiling** is used to break chunks of work up for workers to reduce serially



Serial Reduce – Add Example

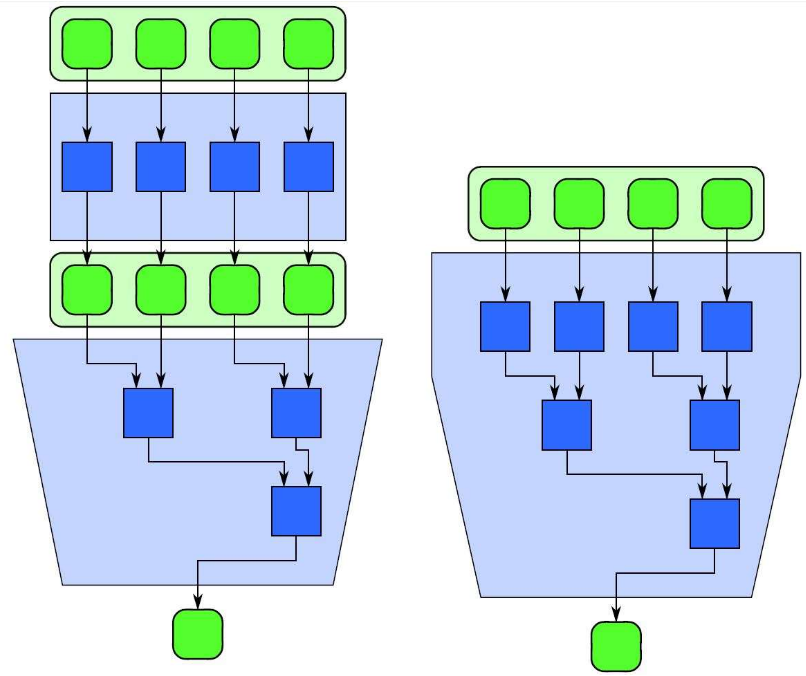


Parallel Reduce – Add Example



Fused Map and Reduce

- We can "fuse" the map and reduce patterns



Precision

- Precision can become a problem with reductions on floating point data
- Different orderings of floating point data can change the reduction value

```
(1.0 + 1.0e10) + -1.0e10
  = 1.0e10 + -1.0e10
  = 0.0
```

but

```
1.0 + (1.0e10 + -1.0e10)
  = 1.0 + 0.0
  = 1.0
```

Reduce Example: Dot Product

- 2 vectors of same length
- Map (*) to multiply the components
- Then reduce with (+) to get the final answer

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n-1} a_i b_i$$

Scan

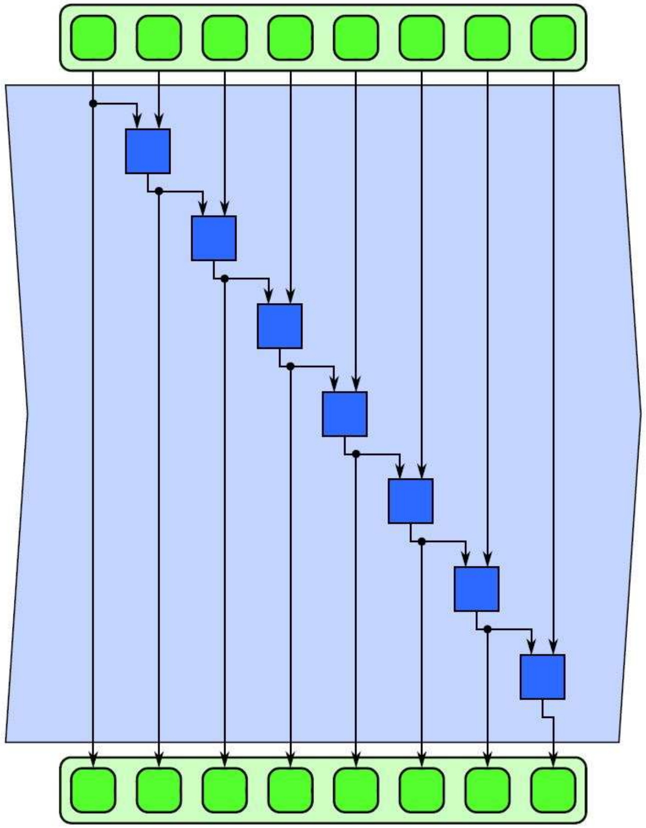
- The **scan** pattern produces partial reductions of input sequence, generates new sequence
- Trickier to parallelize than reduce
- Inclusive scan vs. exclusive scan
  - Inclusive scan: includes current element in partial reduction
  - Exclusive scan: excludes current element in partial reduction, partial reduction is of all prior elements prior to current element

Scan – Example Uses

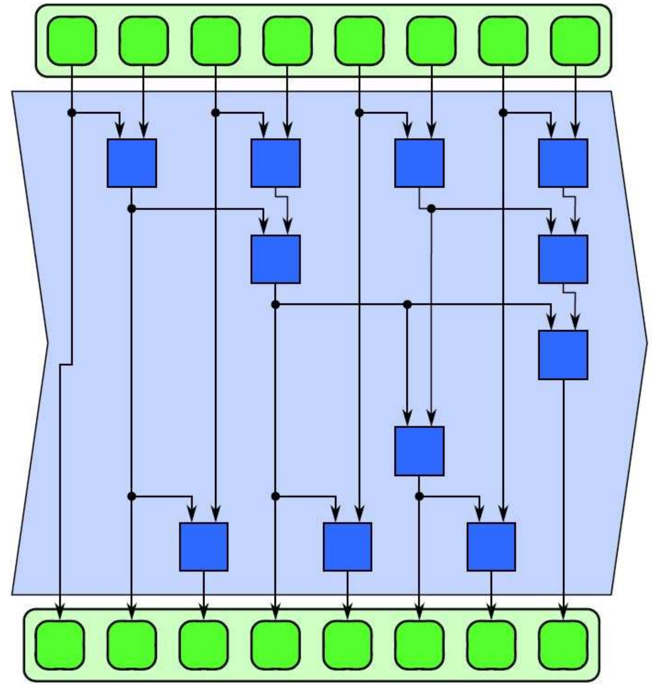
- Lexical comparison of strings – e.g., determine that “strategy” should appear before “stratification” in a dictionary
- Add multi-precision numbers (those that cannot be represented in a single machine *word*)
- Evaluate polynomials
- Implement radix sort or quicksort
- Delete marked elements in an array
- Dynamically allocate processors
- Lexical analysis – parsing programs into tokens
- Searching for regular expressions
- Labeling components in 2-D images
- Some tree algorithms – e.g., finding the depth of every vertex in a tree

Serial vs. Parallel Scan

Serial Scan

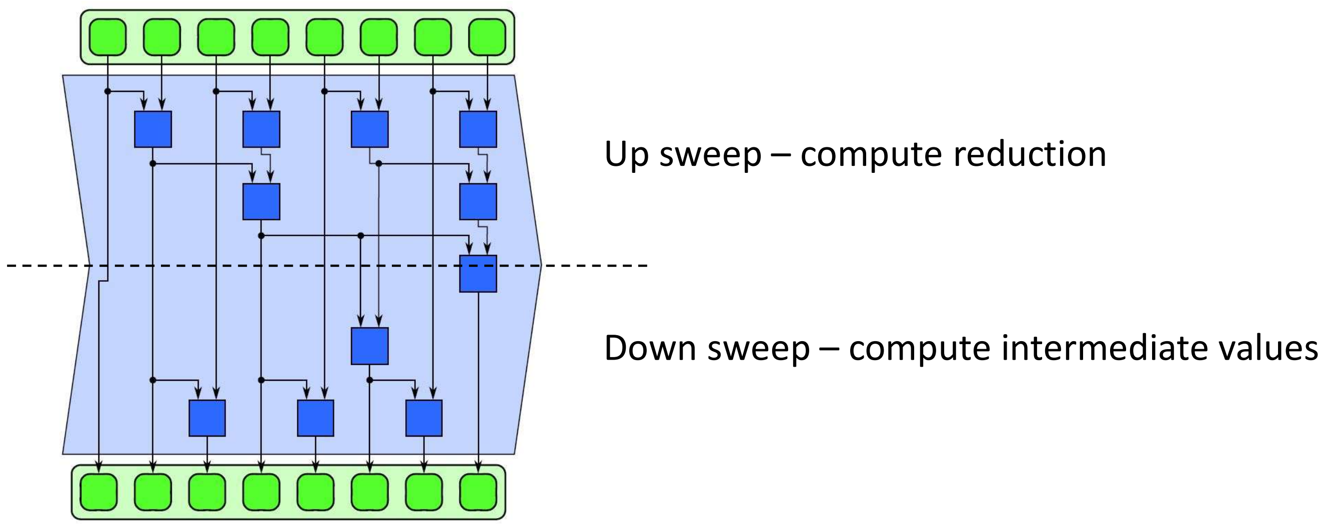


Parallel Scan

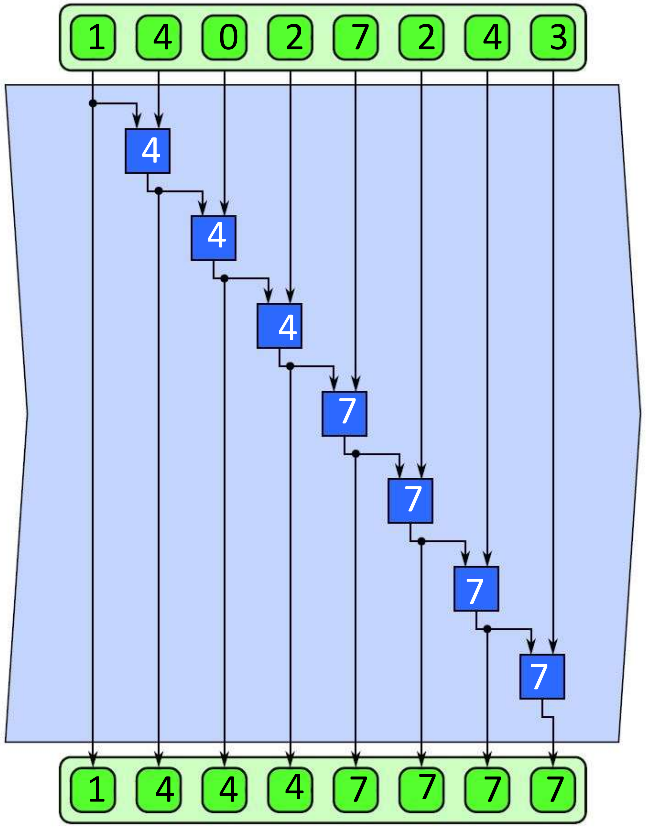


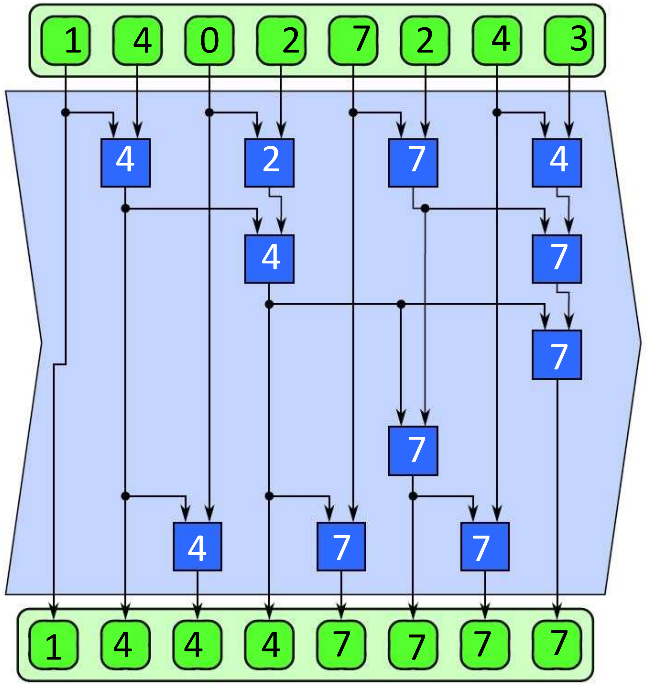
Scan

- One algorithm for parallelizing scan is to perform an “up sweep” and a “down sweep”
- Reduce the input on the up sweep
- The down sweep produces the intermediate results



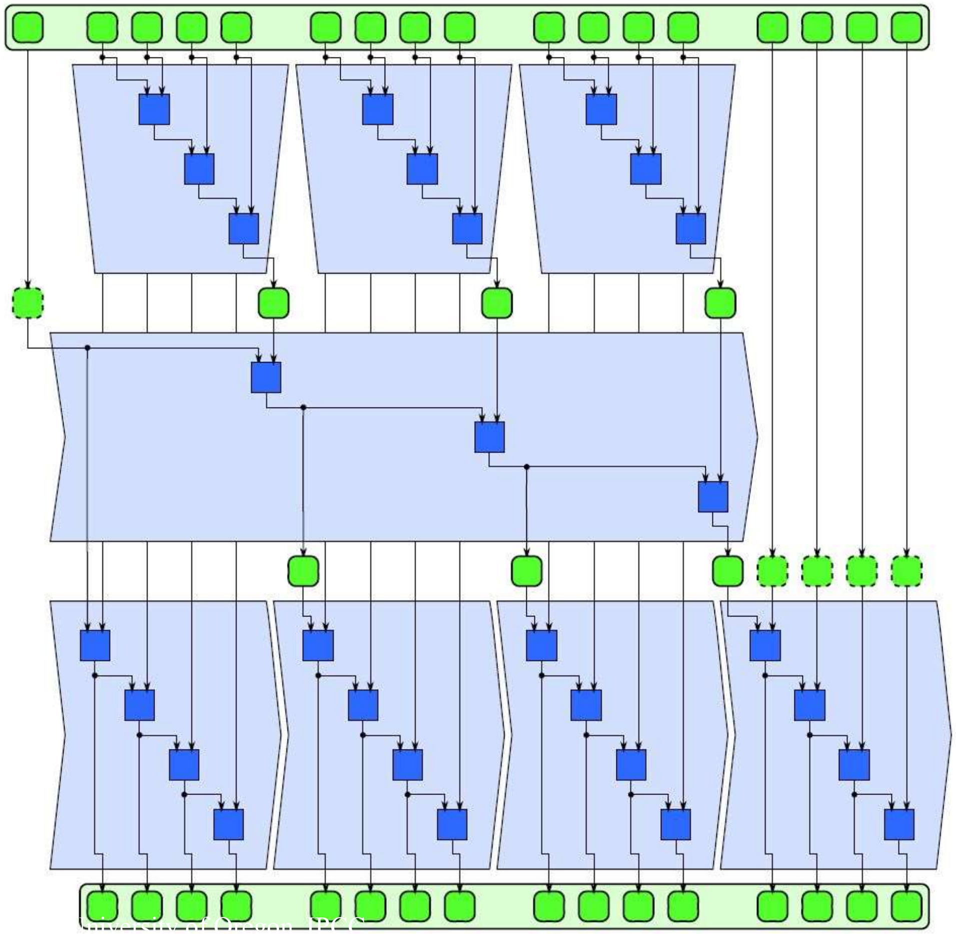
Scan – Maximum Example





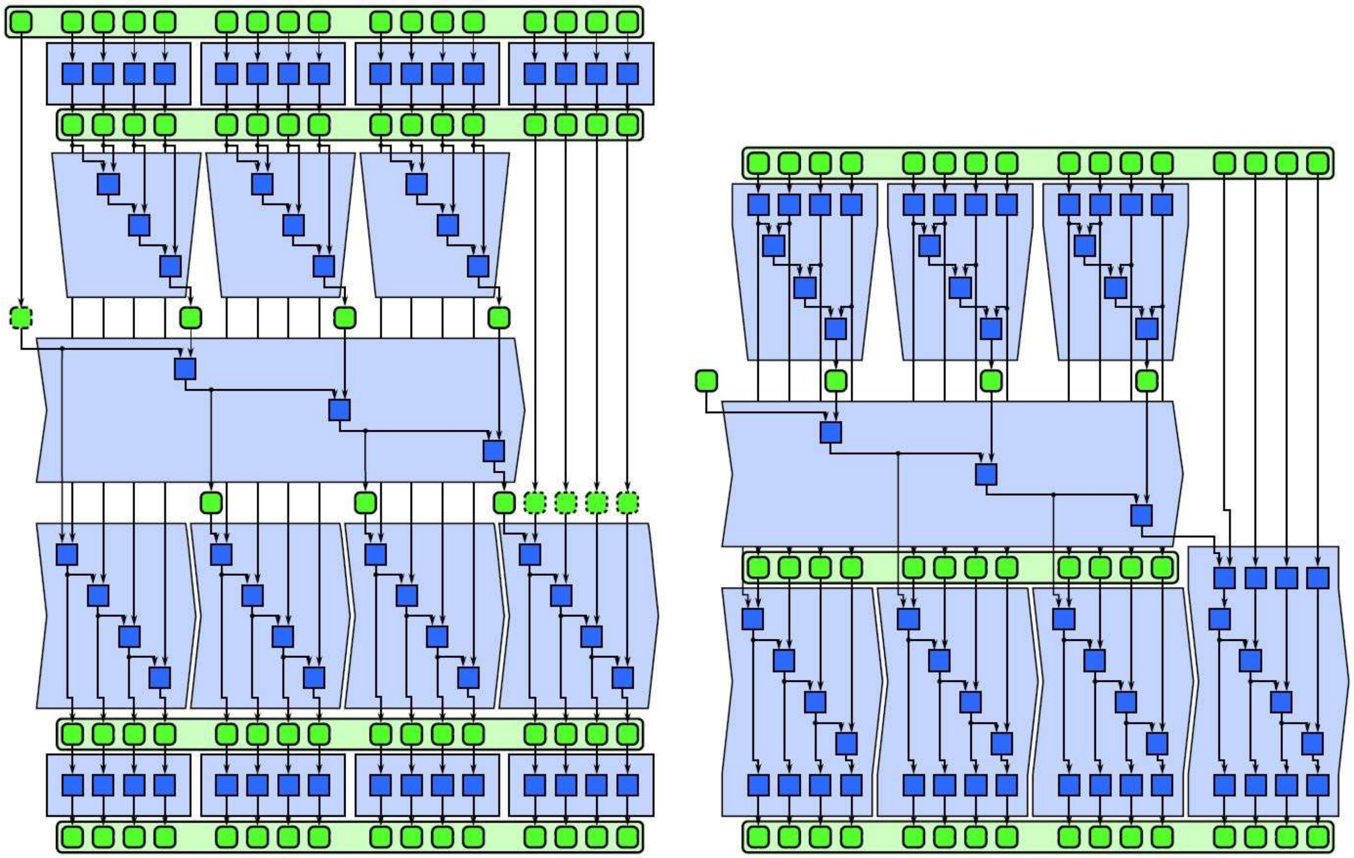
Tiled Scan

Three phase scan with tiling



Fused Map and Scan

- We can also fuse the **map** pattern with the **scan** pattern



Data Movement

- Performance is often more limited by data movement than by computation
  - Transferring data across memory layers is costly
    - locality is important to minimize data access times
    - data organization and layout can impact this
  - Transferring data across networks can take many cycles
    - attempting to minimize the # messages and overhead is important
  - Data movement also costs more in power
- For “data intensive” application, it is a good idea to design the data movement first
  - Design the computation around the data movements
  - Applications such as search and sorting are all about data movement and reorganization

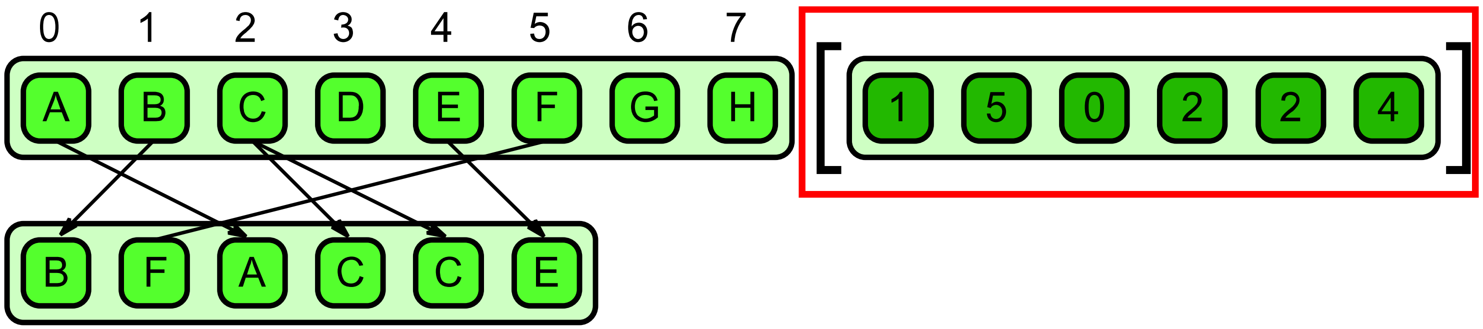
Parallel Data Reorganization

- Remember we are looking to do things in parallel
- How to be faster than the sequential algorithm?
- Similar consistency issues arise as when dealing with computation parallelism
- Here we are concerned more with parallel data movement and management issues
- Might involve the creation of additional data structures (e.g., for holding intermediate data)

Gather Pattern

- Gather pattern creates an (output) collection of data by reading from another (source) data collection
  - Given a collection of (ordered) indices
  - Read data from the source collection at each index
  - Write data to the output collection in index order
- Transfers from source collection to output collection
  - Element type of output collection is the same as the source
  - Shape of the output collection is that of the index collection
    - same dimensionality
- Can be considered a combination of map and random serial read operations
  - Essentially does a number of random reads in parallel

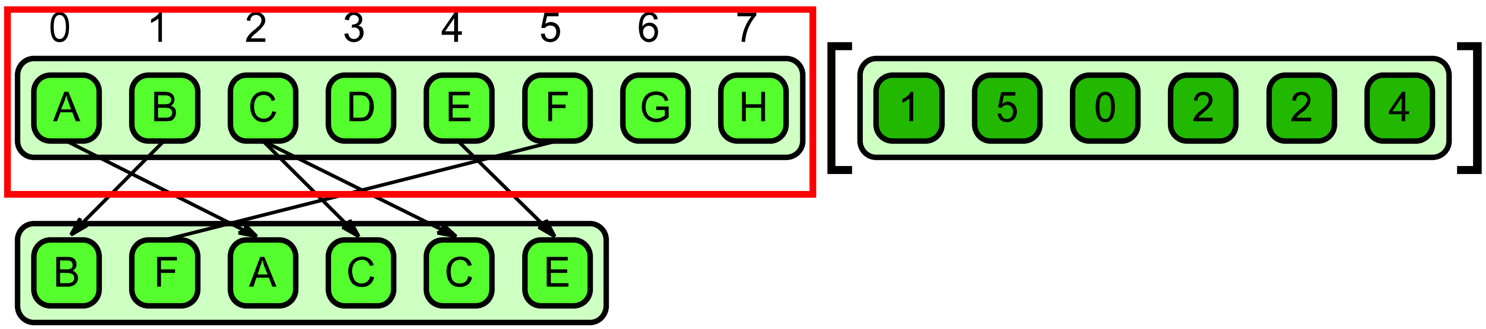
Gather: Defined



Given a **collection of read locations**

- address or array indices

Gather: Defined

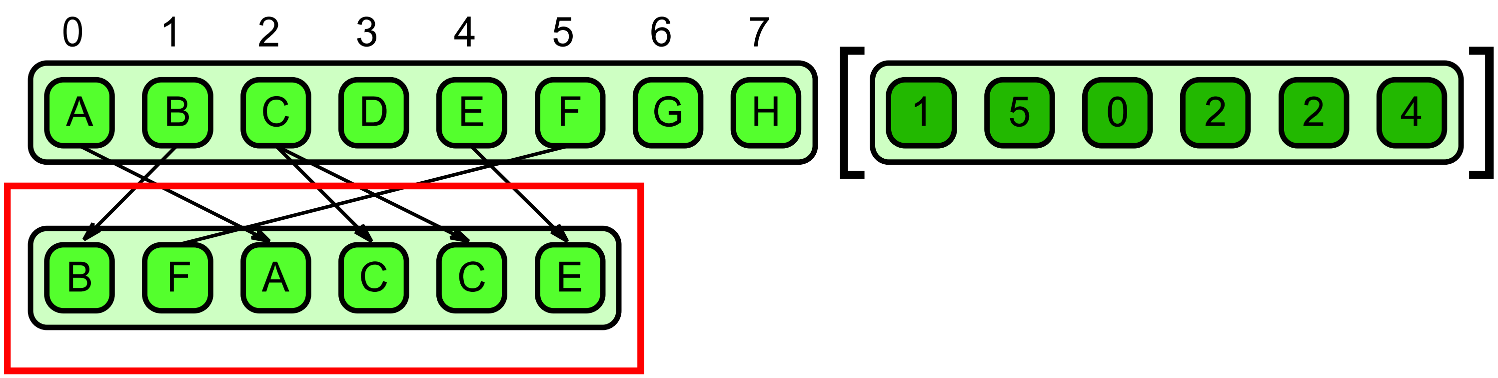


Given a collection of read locations

- address or array indices

and a **source array**

Gather: Defined



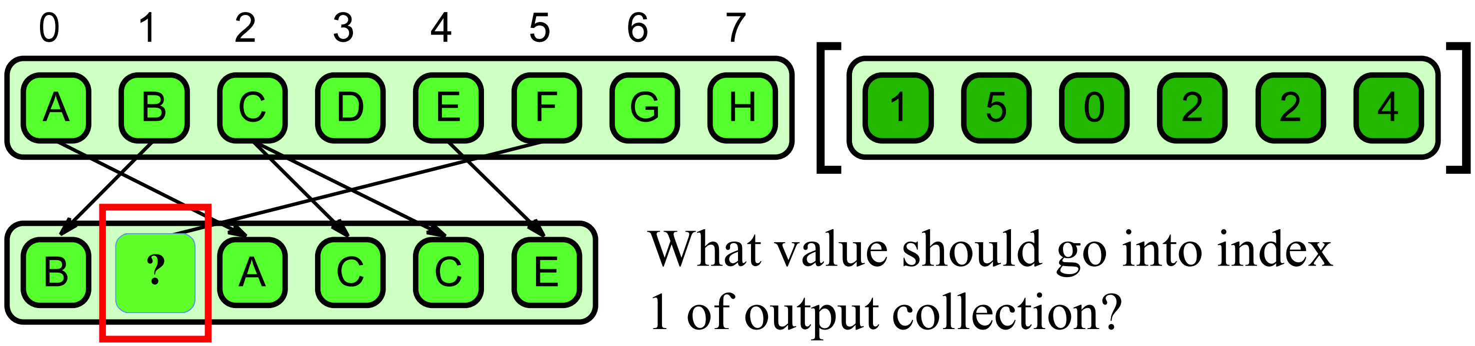
Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



What value should go into index 1 of output collection?

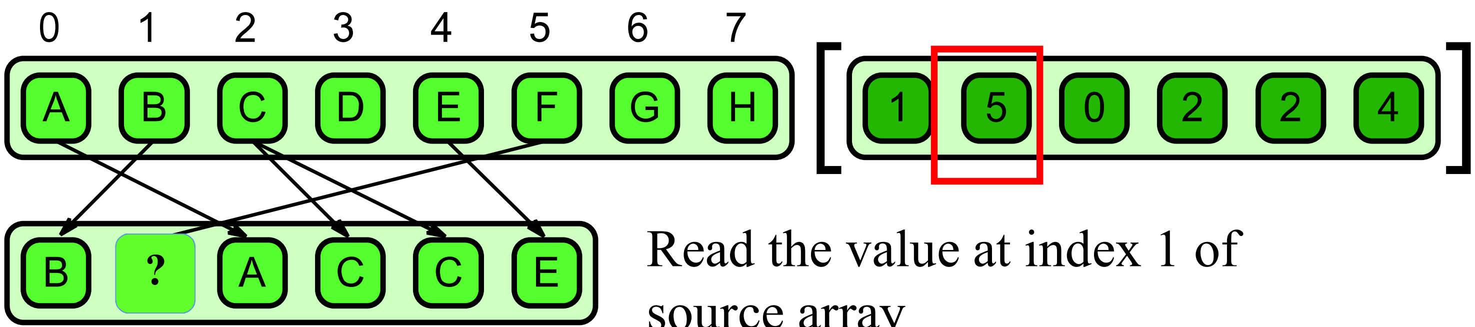
Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



Read the value at index 1 of source array

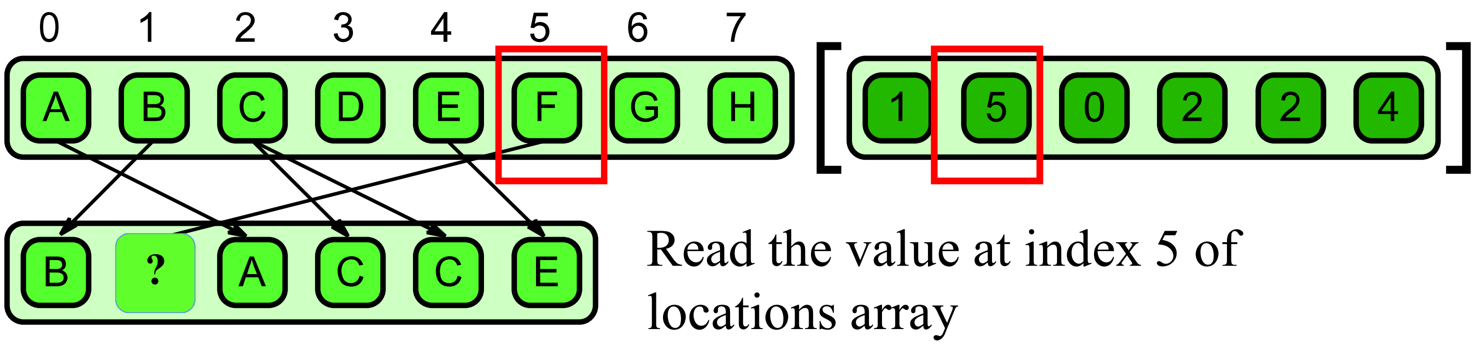
Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



Read the value at index 5 of locations array

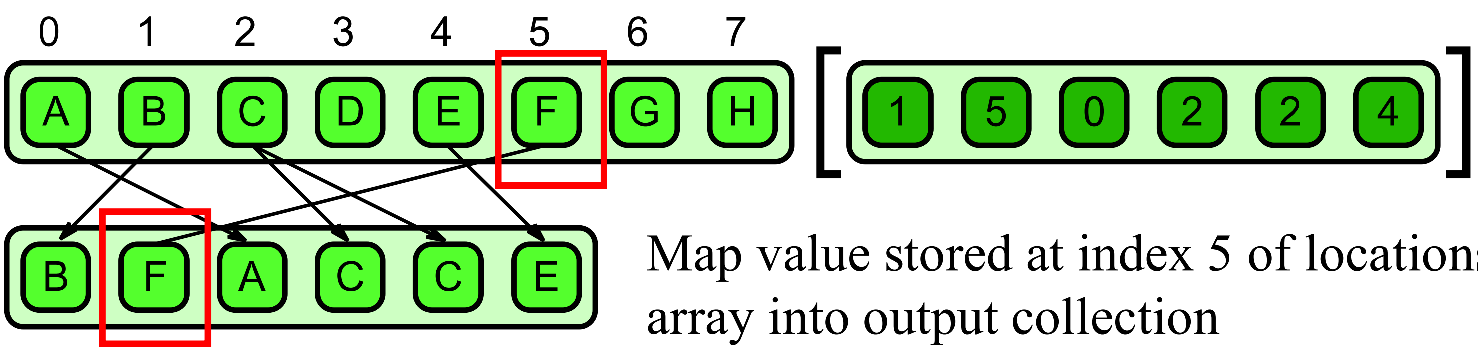
Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined



Map value stored at index 5 of locations array into output collection

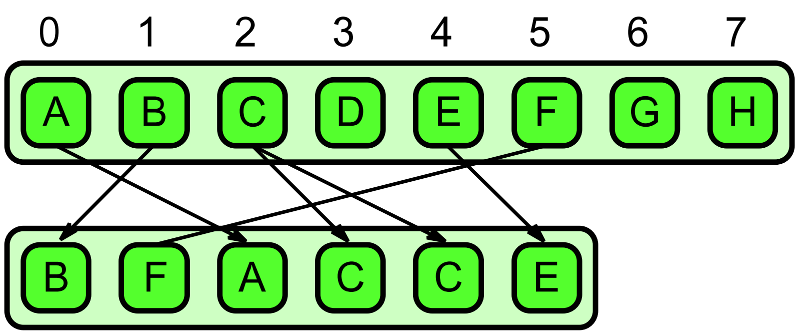
Given a collection of read locations

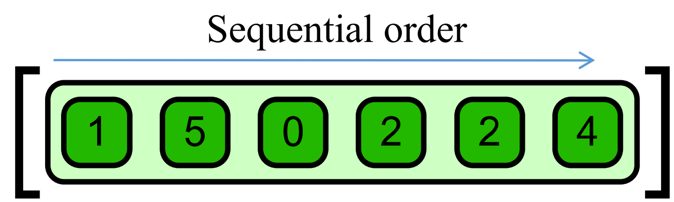
- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Gather: Defined





Where is the parallelism?

Given a collection of read locations

- address or array indices

and a source array

gather all the data from the source array at the given locations and places them into an **output collection**

Quiz 1

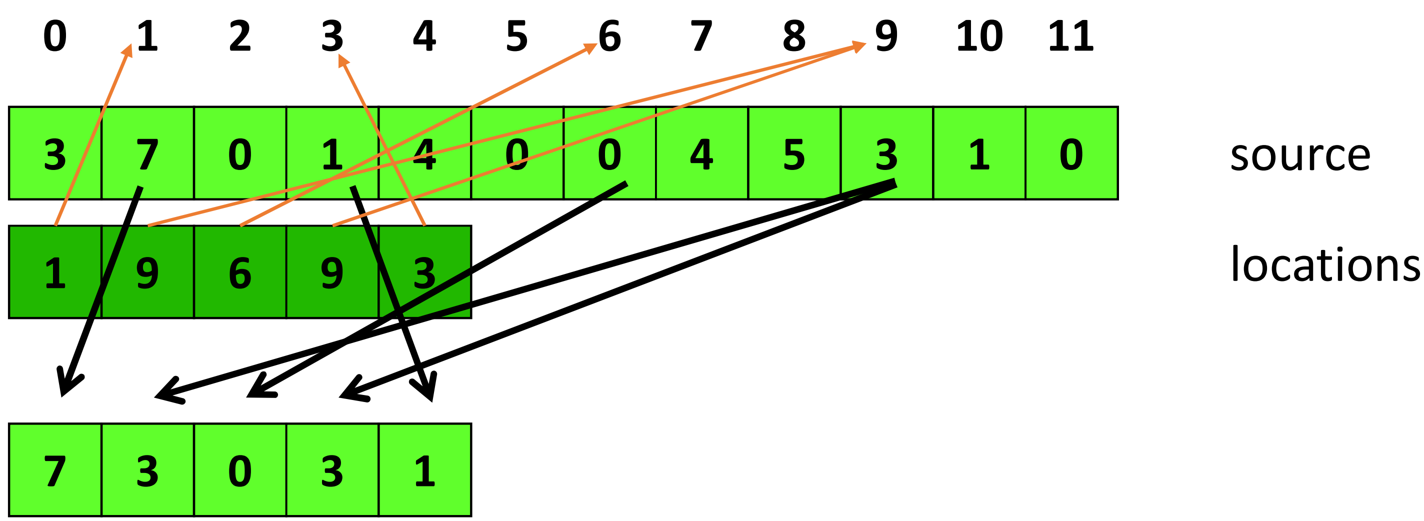
Given the following source and locations array, use a gather to determine what values should go into the output collection:

0	1	2	3	4	5	6	7	8	9	10	11	
3	7	0	1	4	0	0	4	5	3	1	0	source
1	9	6	9	3								locations

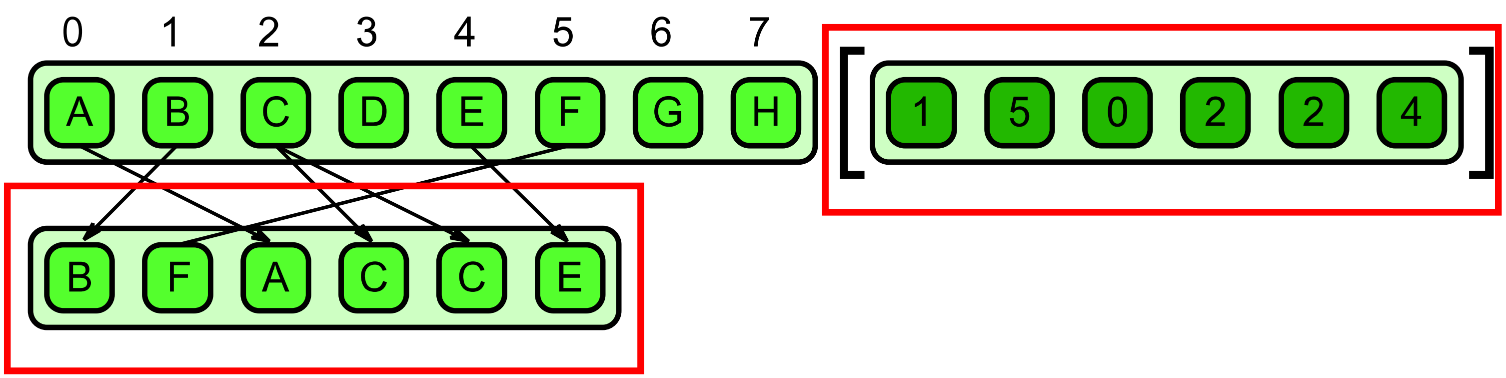
?	?	?	?	?

Quiz 1 Answer

Given the following source and locations array, use a gather to determine what values should go into the output collection:

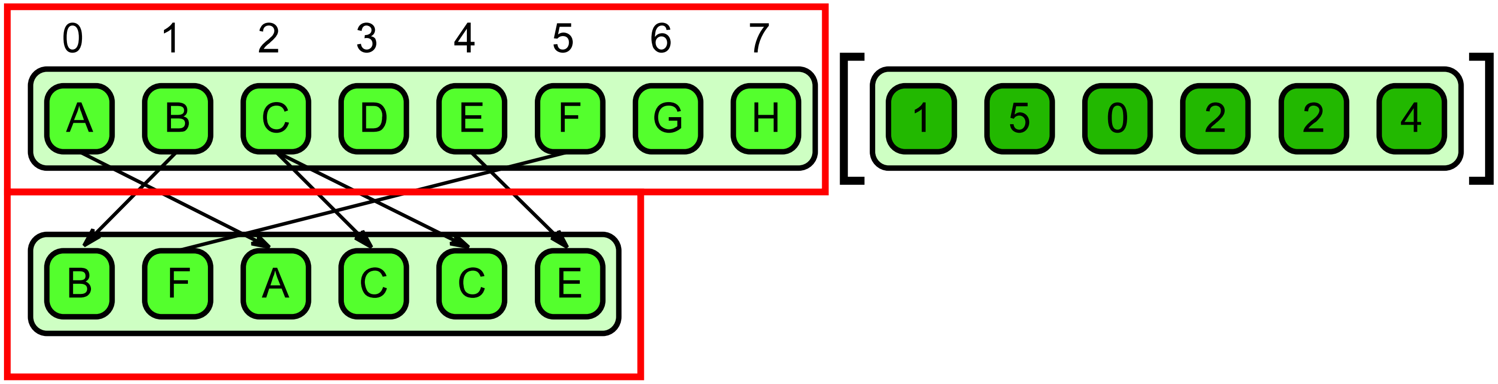


Gather: Array Size



- Output data collection has the same number of elements as the number of indices in the index collection
  - Same dimensionality

Gather: Array Type



- Output data collection has the same number of elements as the number of indices in the index collection
- Elements of the output collection are the same type as the input data collection

Gather: Serial Implementation

```
template<typename Data, typename Idx>
void gather(
    size_t n, // number of elements in data collection
    size_t m, // number of elements in index collection
    Data a[], // input data collection (n elements)
    Data A[], // output data collection (m elements)
    Idx idx[] // input index collection (m elements)
) {
    for (size_t i = 0; i < m; ++i) {
        size_t j = idx[i]; // get ith index
        assert(0 <= j && j < n); // check array bounds
        A[i] = a[j]; // perform random read
    }
}
```

Serial implementation of gather in pseudocode

Gather: Serial Implementation

```
template<typename Data, typename Idx>
void gather(
    size_t n,  // number of elements in data collection
    size_t m,  // number of elements in index collection
    Data a[],  // input data collection (n elements)
    Data A[],  // output data collection (m elements)
    Idx idx[]  // input index collection (m elements)
) {
    for (size_t i = 0; i < m; ++i) {
        size_t j = idx[i];  // get ith index
        assert(0 <= j && j < n);  // check array bounds
        A[i] = a[j];  // perform random read
    }
}
```

Serial implementation of gather in pseudocode

Do you see opportunities for parallelism?

Gather: Serial Implementation

```
template<typename Data, typename Idx>
void gather(
    size_t n,  // number of elements in data collection
    size_t m,  // number of elements in index collection
    Data a[],  // input data collection (n elements)
    Data A[],  // output data collection (m elements)
    Idx idx[]  // input index collection (m elements)
) {
    for (size_t i = 0; i < m; ++i) {
        size_t j = idx[i]; // get ith index
        assert(0 <= j && j < n); // check array bounds
        A[i] = a[j]; // perform random read
    }
}
```

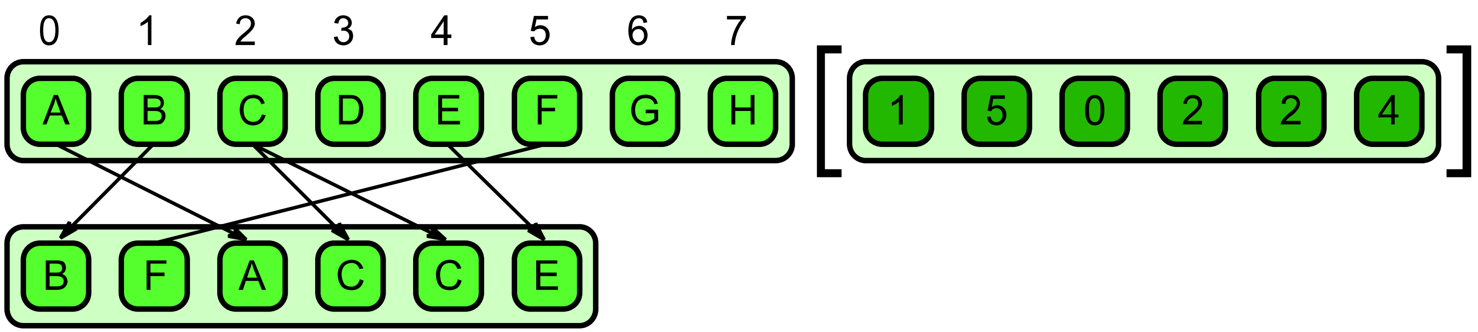
Parallelize over for loop to perform random read

Serial implementation of gather in pseudocode

Are there any conflicts that arise?

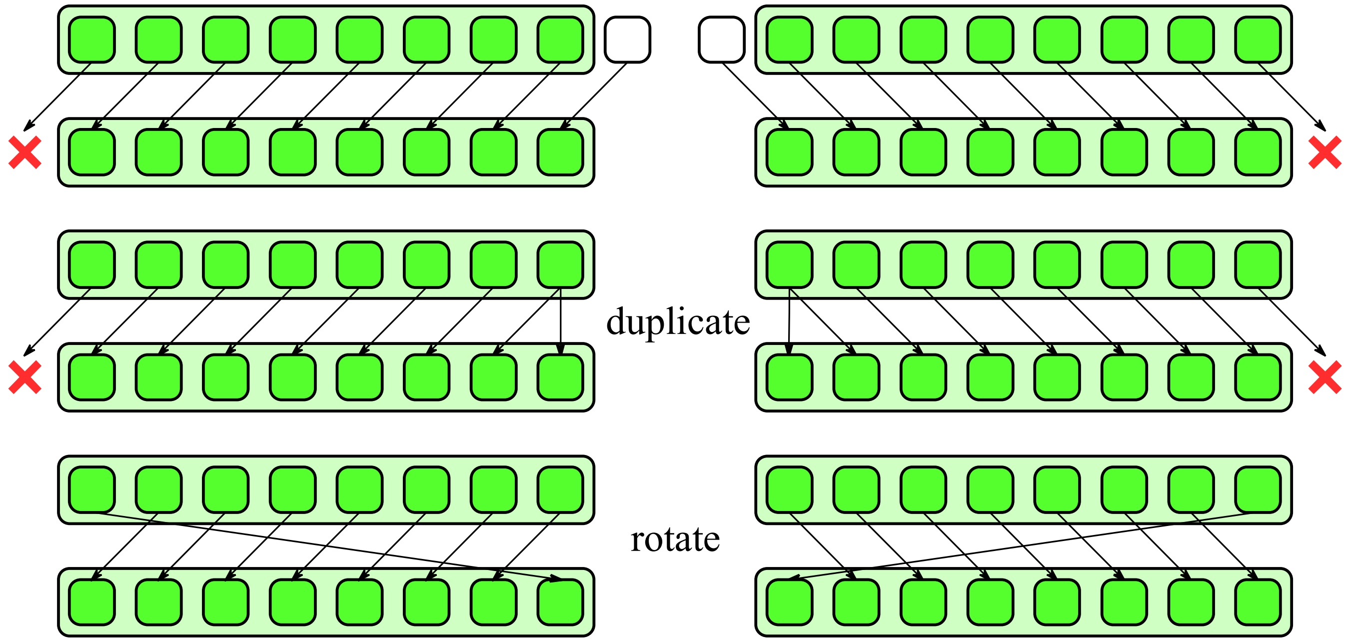
Gather: Defined (parallel perspective)

- Results from the combination of a map with a random read



- Simple pattern, but with many special cases that make the implementation more efficient

Special Case of Gather: Shifts

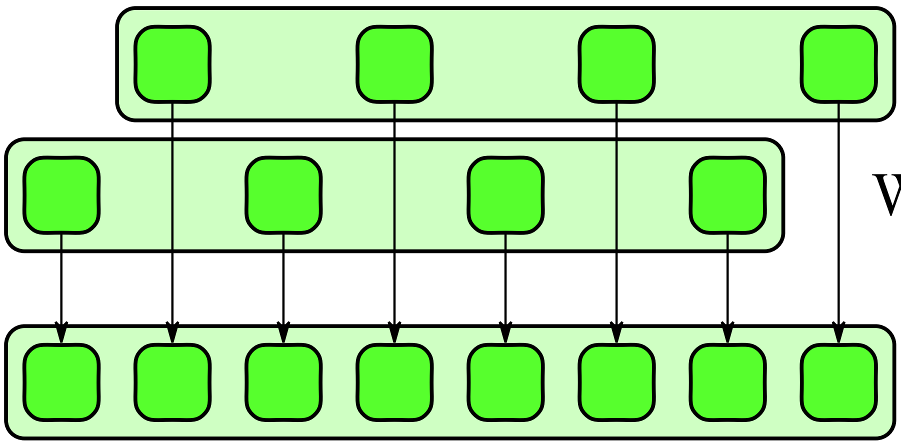


- Moves data to the left or right in memory
- Data accesses are offset by fixed distances

More about Shifts

- Regular data movement
- Variants from how boundary conditions handled
  - Requires “out of bounds” data at edge of the array
  - Options: default value, duplicate, rotate
- Shifts can be handled efficiently with vector instructions because of regularity
  - Shift multiple data elements at the same time
- Shifts can also take advantage of good data locality

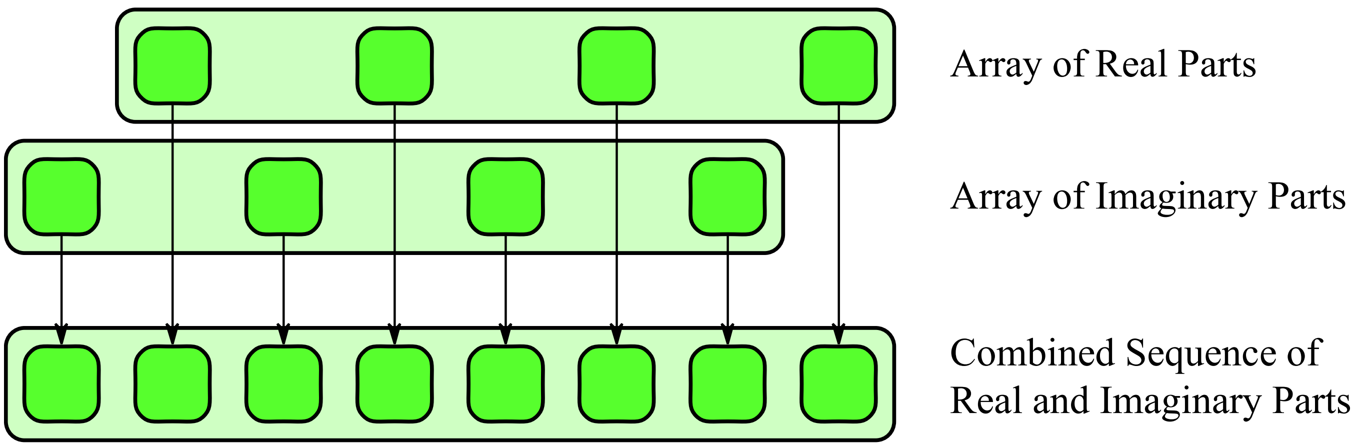
Special Case of Gather: Zip



Where is the parallelism?

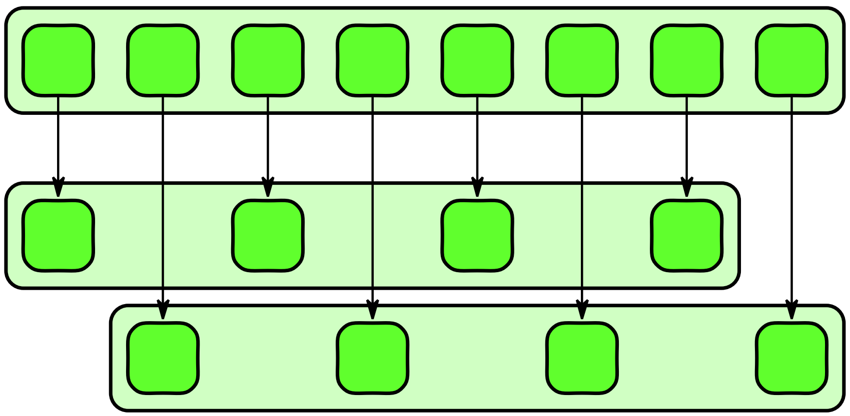
- Function is to interleaves data (like a zipper)

Zip Example



- Given two separate arrays of real parts and imaginary parts
- Use zip to combine them into a sequence of real and imaginary pairs

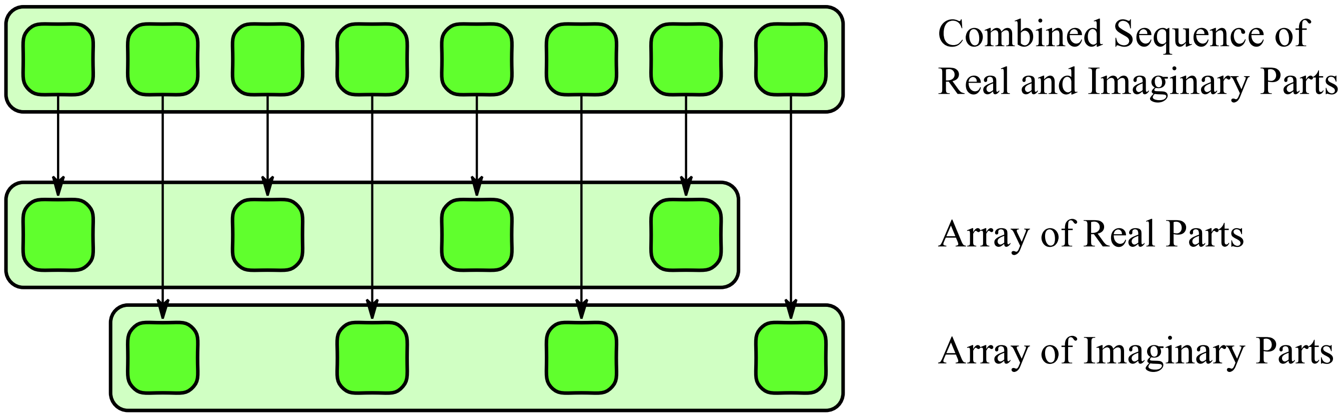
Special Case of Gather: Unzip



Where is the parallelism?

- Reverses a zip
- Extracts sub-arrays at certain offsets and strides from an input array

Unzip Example



- Given a sequence of complex numbers organized as pairs
- Use unzip to extract real and imaginary parts into separate arrays

Gather vs. Scatter

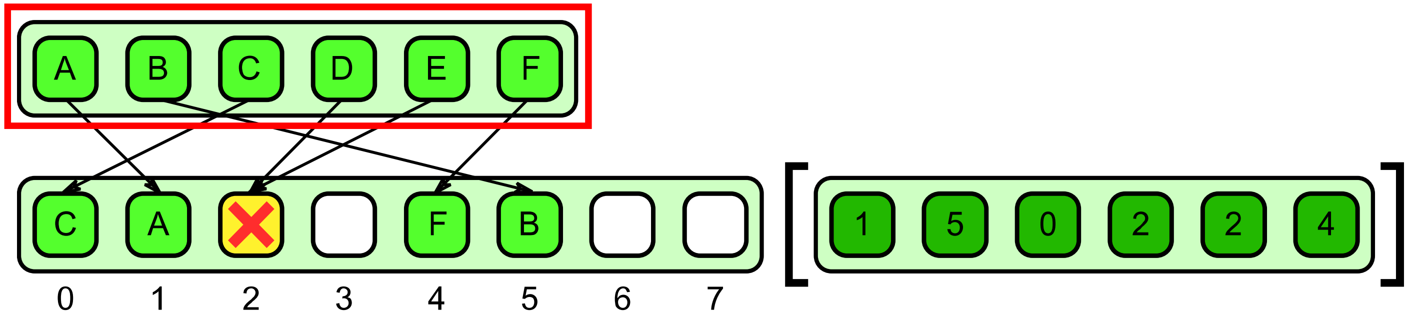
Gather

- Combination of map with random **reads**
- Read locations provided as input

Scatter

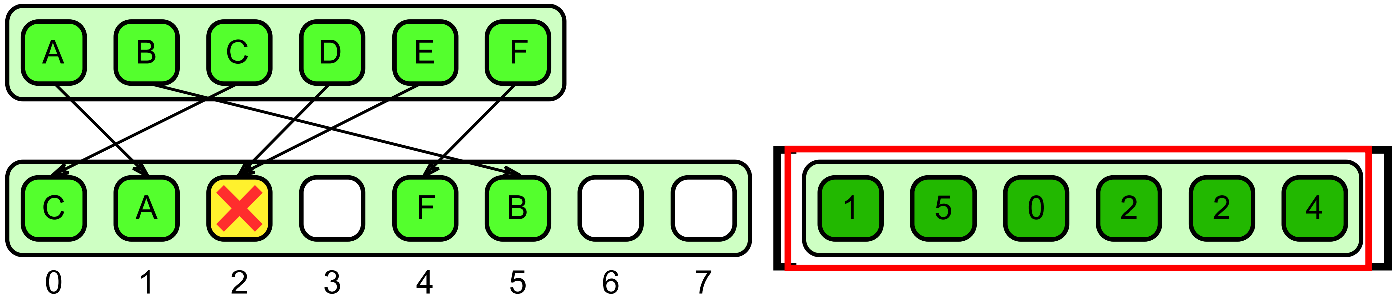
- Combination of map with random **writes**
- Write locations provided as input
- Race conditions … Why?

Scatter: Defined



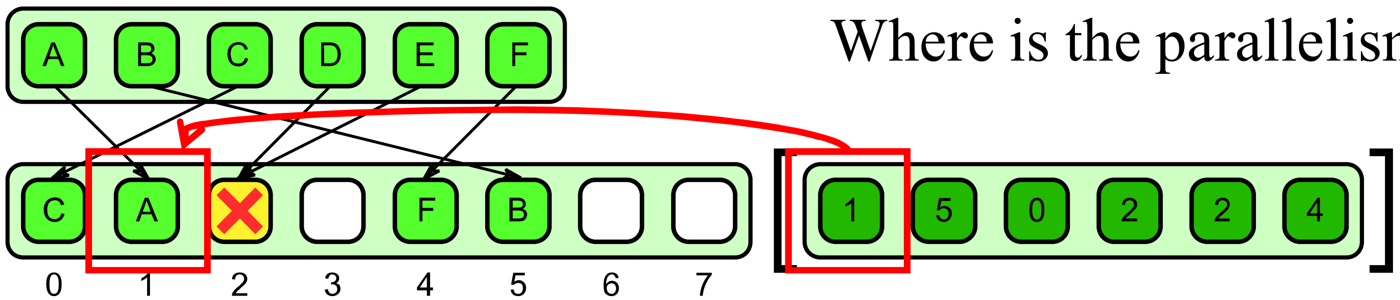
Given a collection of **input data**

Scatter: Defined



Given a collection of input data
and a collection of **write locations**

Scatter: Defined



Where is the parallelism?

Given a collection of input data
and a collection of write locations
scatter data to the **output collection**

Problems?

Does the output collection have to be larger in size?

Quiz 2

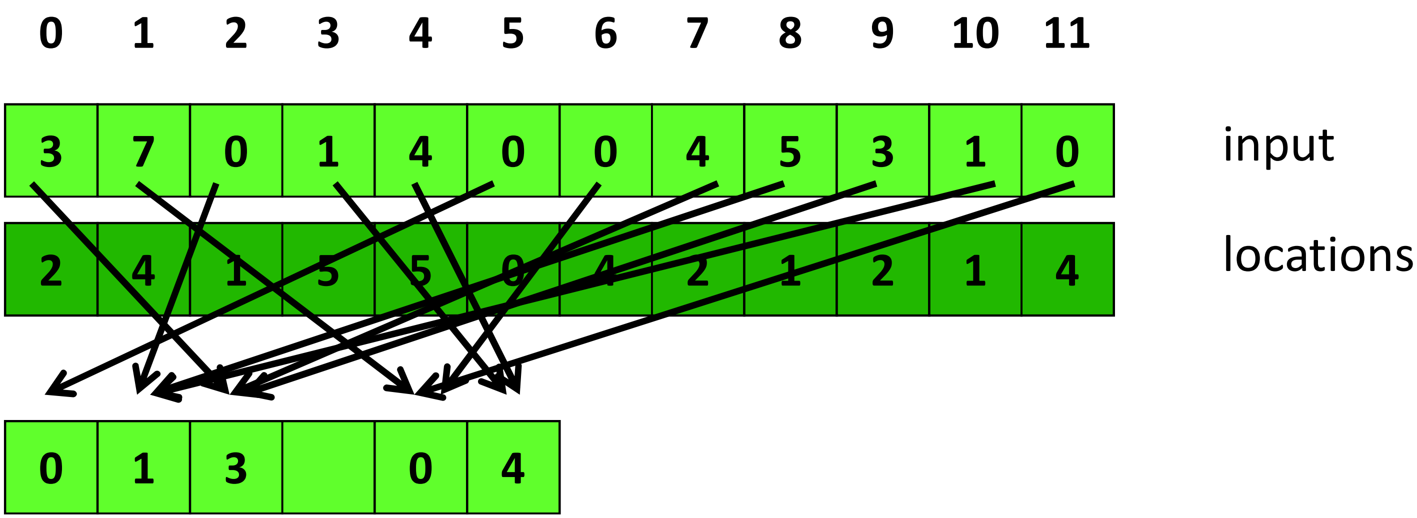
Given the following input and locations array, what values should go into the output collection:

	0	1	2	3	4	5	6	7	8	9	10	11
input	3	7	0	1	4	0	0	4	5	3	1	0
locations	2	4	1	5	5	0	4	2	1	2	1	4

?	?	?		?	?

Quiz 2 Answer

Given the following input and locations array, what values should go into the output collection:



Scatter: Serial Implementation

```
template<typename Data, typename Idx>
void scatter(
    size_t n, // number of elements in output data collection
    size_t m, // number of elements in input data and index collection
    Data a[], // input data collection (m elements)
    Data A[], // output data collection (n elements)
    Idx idx[] // input index collection (m elements)
) {
    for (size_t i = 0; i < m; ++i) {
        size_t j = idx[i]; // get ith index
        assert(0 <= j && j < n); // check output array bounds
        A[j] = a[i]; // perform random write
    }
}
```

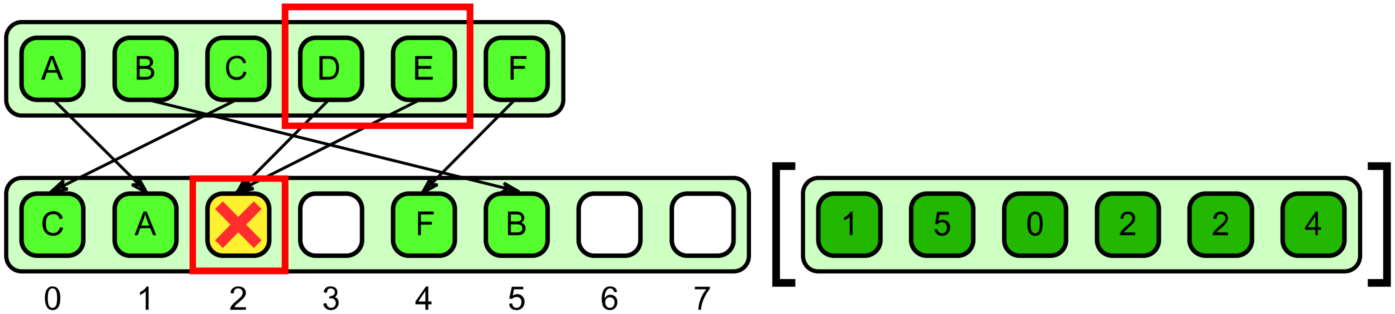
Parallelize over for loop to perform random write

Serial implementation of scatter in pseudocode

Scatter: Defined

- Results from the combination of a map with a random write
- Writes to the same location are possible
- Parallel writes to the same location are **collisions**

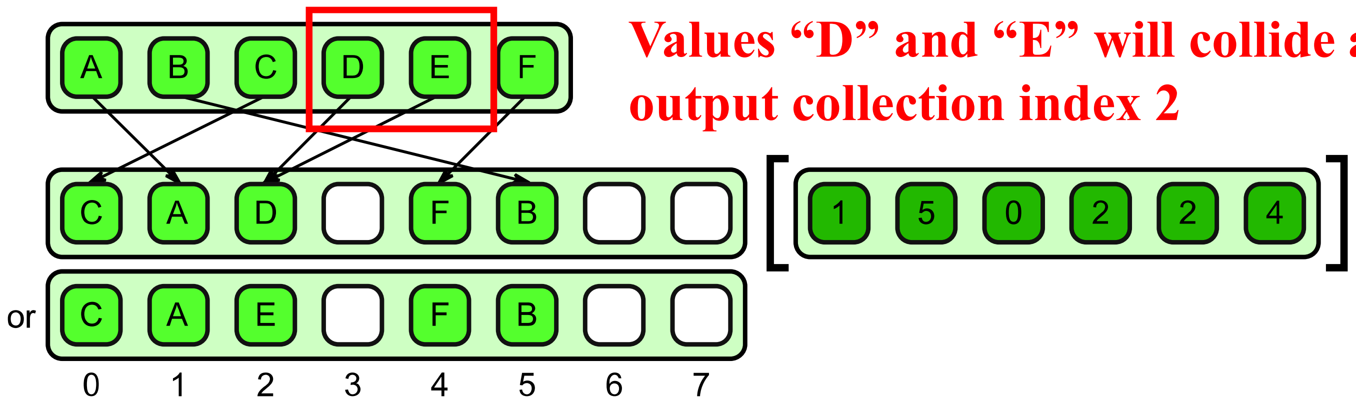
Scatter: Race Conditions



Given a collection of input data
and a collection of write locations
scatter data to the output collection

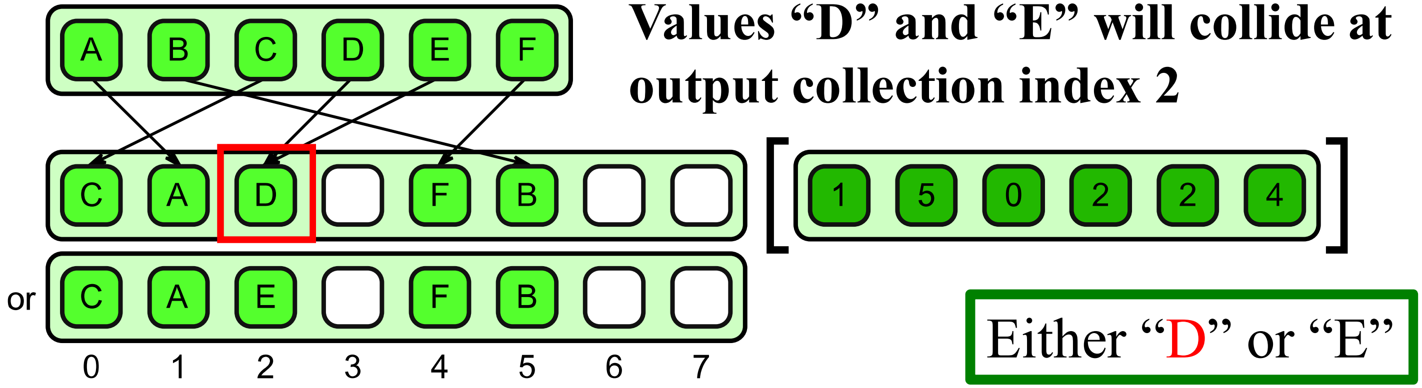
Race Condition: Two (or more) values being written to the same location in output collection. Result is undefined unless enforce rules. **Need rules to resolve collisions!**

Collision Resolution: Atomic Scatter



- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety

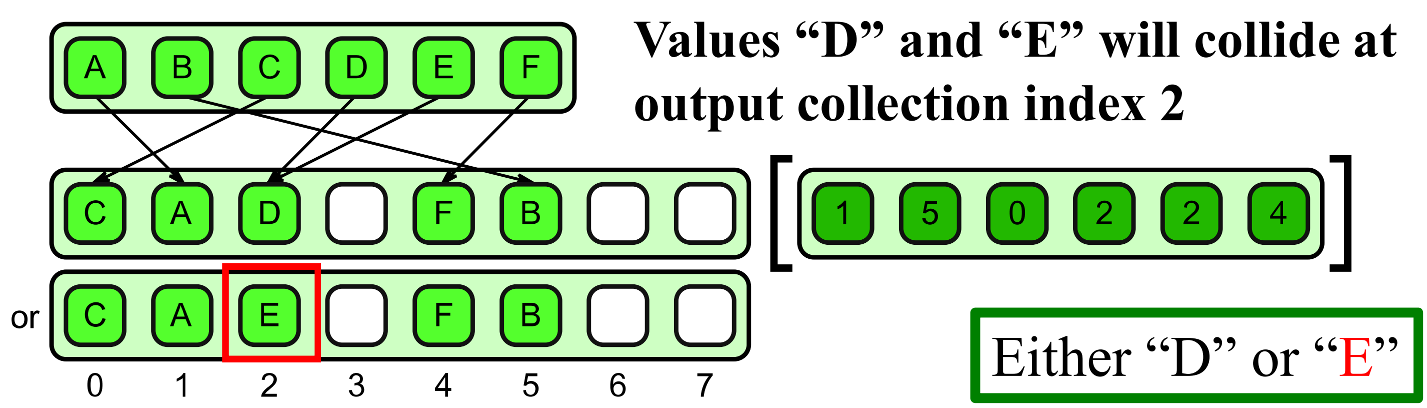
Collision Resolution: Atomic Scatter



Values "D" and "E" will collide at output collection index 2

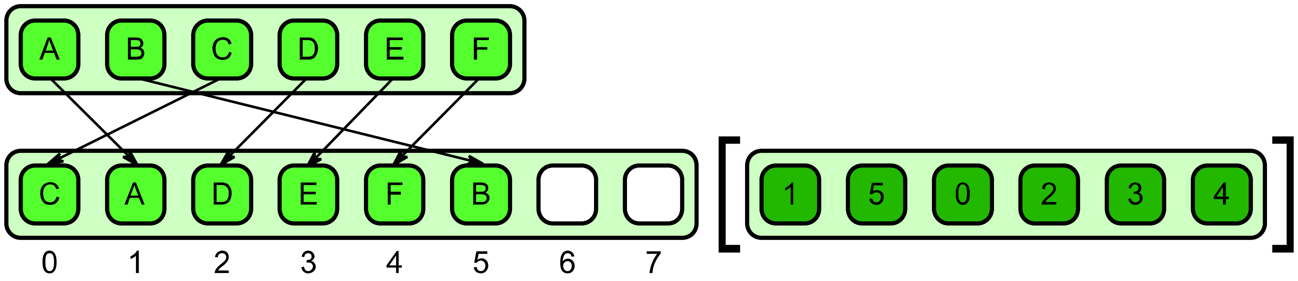
- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
- No rule determines which of the input items will be retained

Collision Resolution: Atomic Scatter



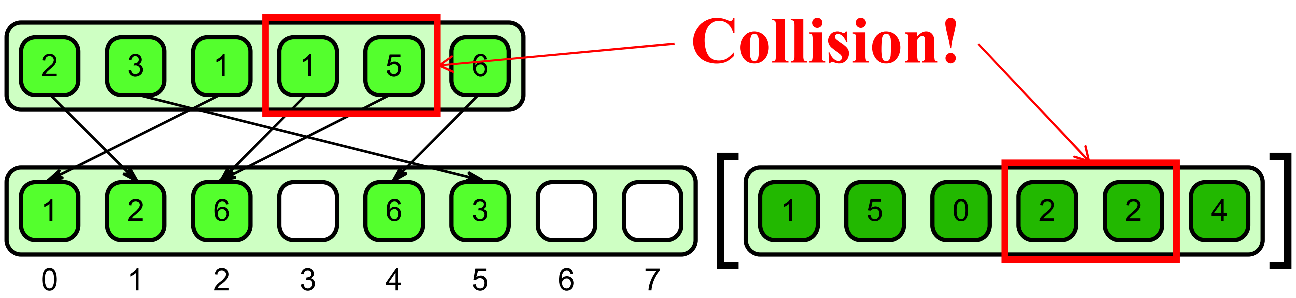
- Non-deterministic approach
- Upon collision, one and only one of the values written to a location will be written in its entirety
- No rule determines which of the input items will be retained

Collision Resolution: Permutation Scatter



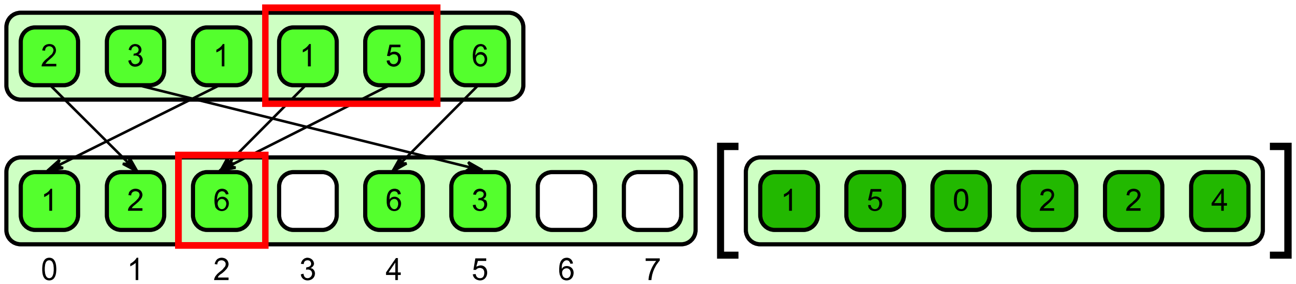
- Pattern simply states that collisions are **illegal**
  - Output is a permutation of the input
- Check for collisions in advance
  → turn scatter into gather
- Examples
  - FFT scrambling, matrix/image transpose, unpacking

Collision Resolution: Merge Scatter



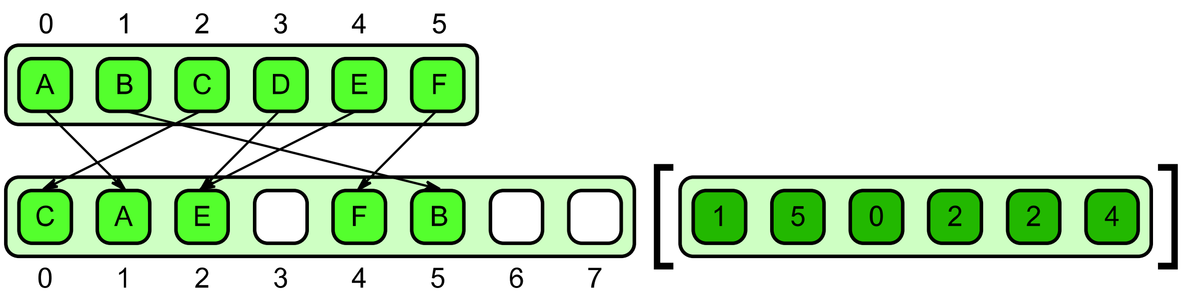
- Associative and commutative operators are provided to merge elements in case of a collision

Collision Resolution: Merge Scatter



- Associative and commutative operators are provided to merge elements in case of a collision
- Use addition as the merge operator
- Both associative and commutative properties are required since scatters to a particular location could occur in any order

Collision Resolution: Priority Scatter

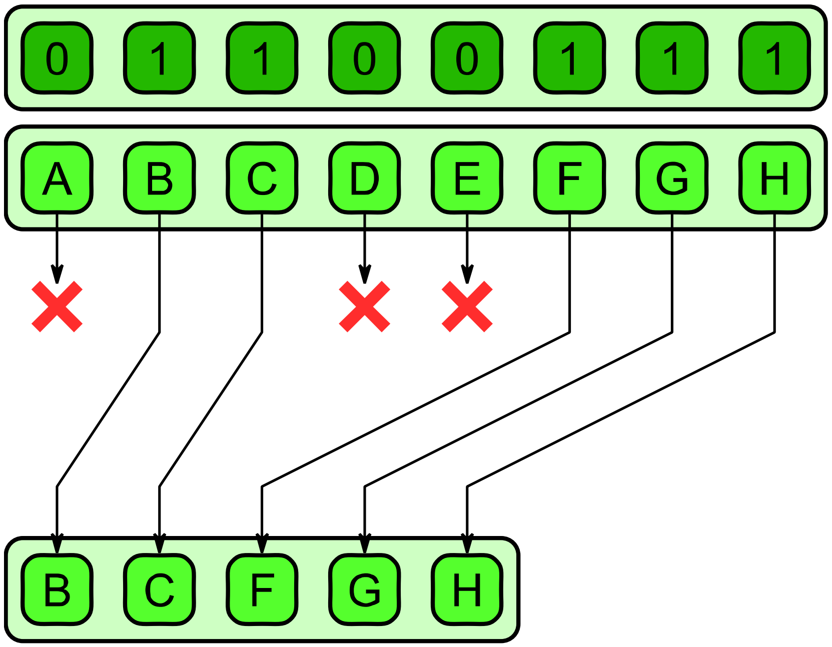


- Every element in the input array is assigned a priority based on its position
- Priority is used to decide which element is written in case of a collision
- Example
  - 3D graphics rendering

Converting Scatter to Gather

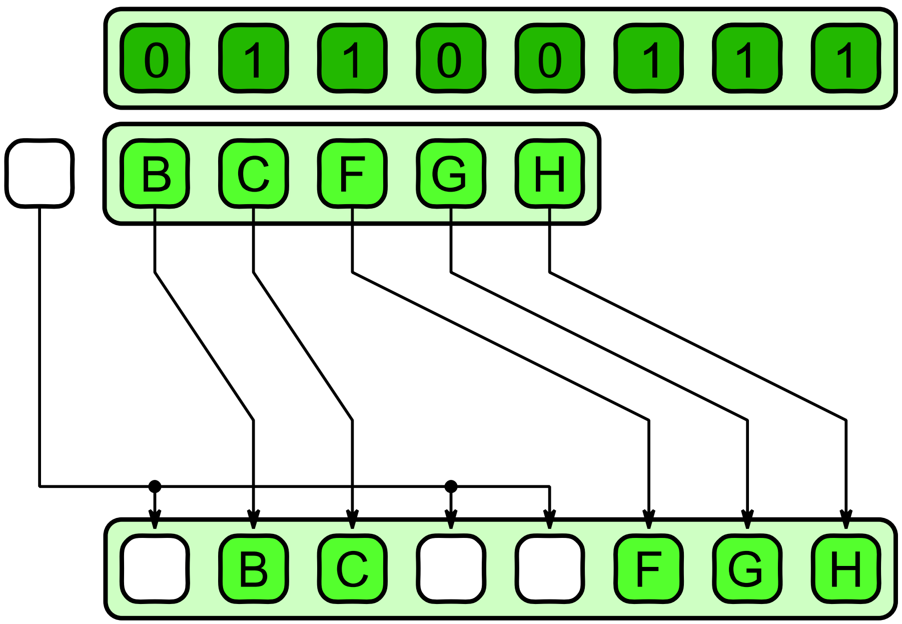
- Scatter is a more expensive than gather
  - Writing has cache line consequences
  - May cause additional reading due to cache conflicts
  - **False sharing** is a problem that arises
    - writes from different cores go to the same cache line
- Can avoid problems if addresses are know "in advance"
  - Allows optimizations to be applied
  - Convert addresses for a scatter into those for a gather
  - Useful if the same pattern of scatter address will be used repeatedly so the cost is amortized

Pack



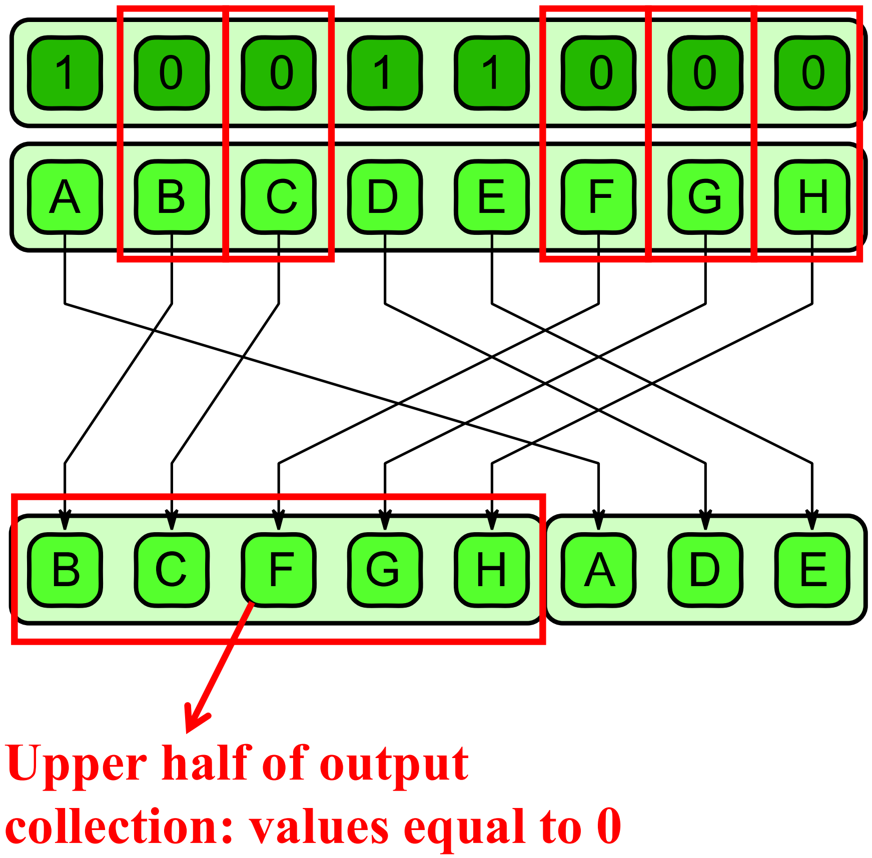
- Used to eliminate unused elements from a collection
- Retained elements are moved so they are contiguous in memory

Unpack



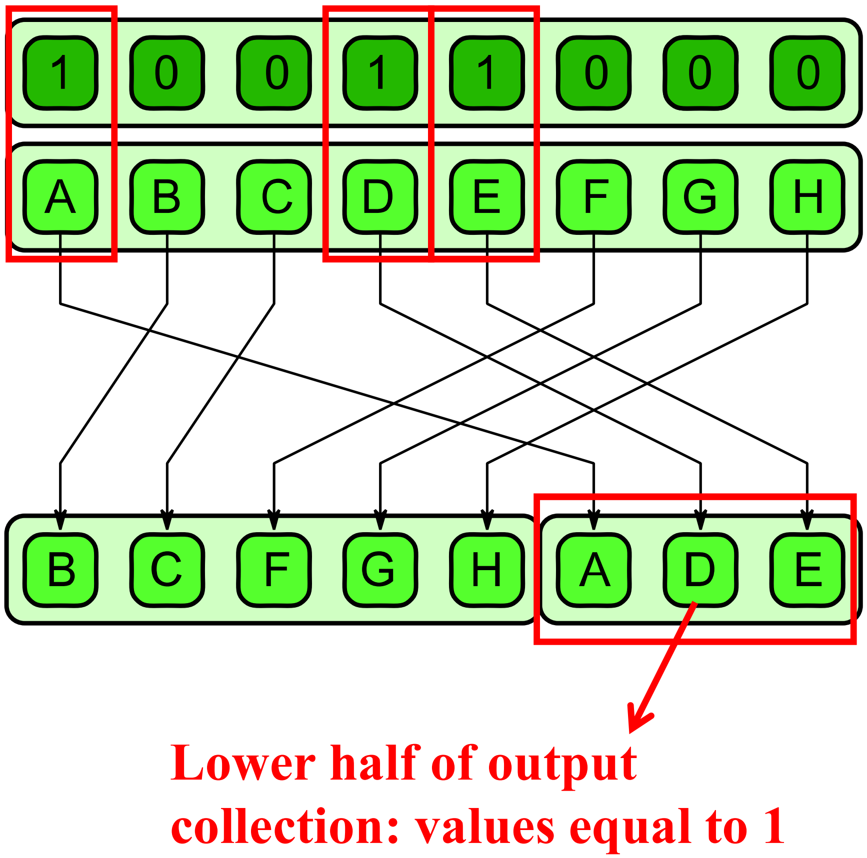
- Inverse of pack operation
- Given the same data on which elements were kept and which were discarded, spread elements back in their original locations

Generalization of Pack: Split



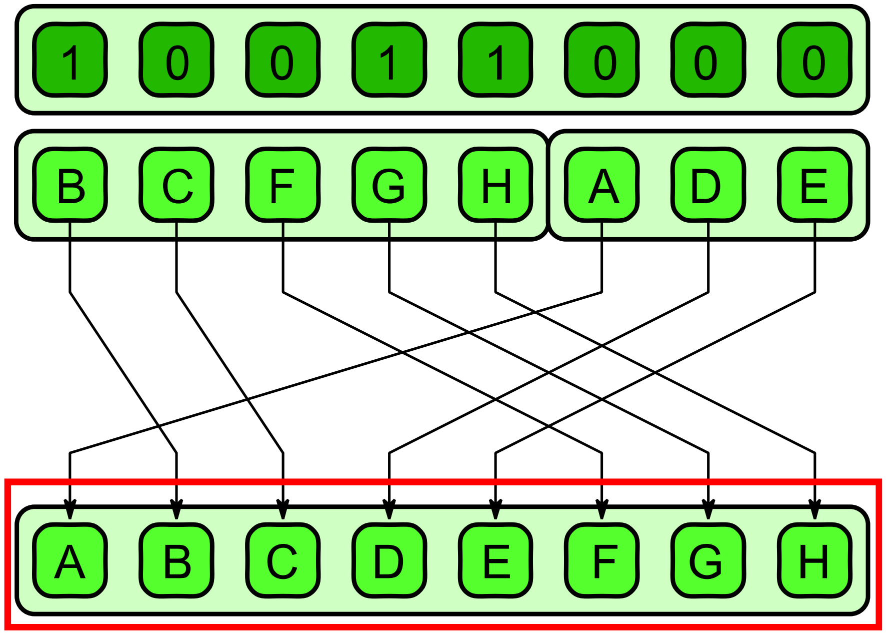
- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack

Generalization of Pack: Split



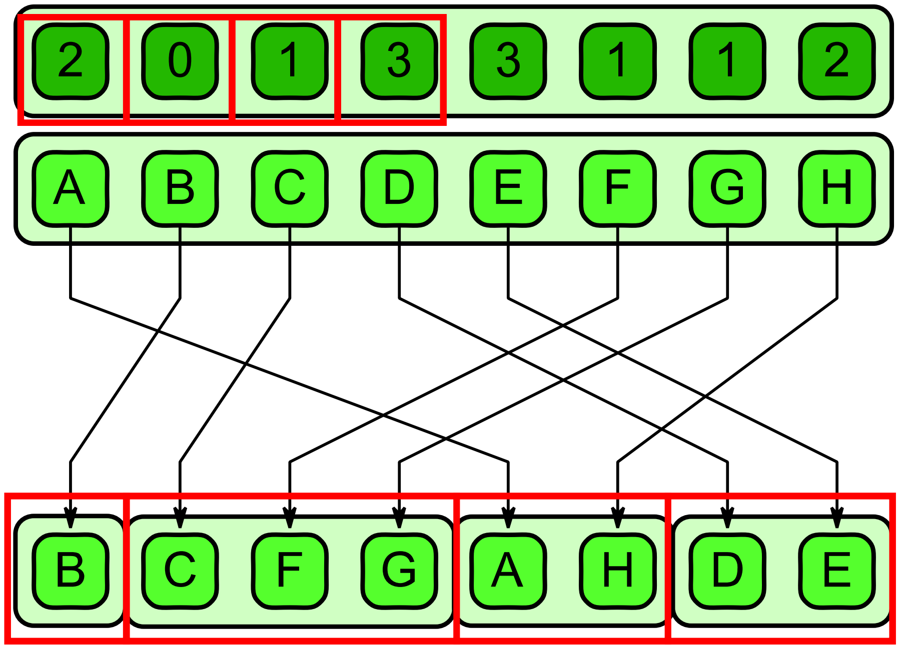
- Generalization of pack pattern
- Elements are moved to upper or lower half of output collection based on some state
- Does not lose information like pack

Generalization of Pack: Unsplit



- Inverse of split
- Creates **output collection** based on original input collection

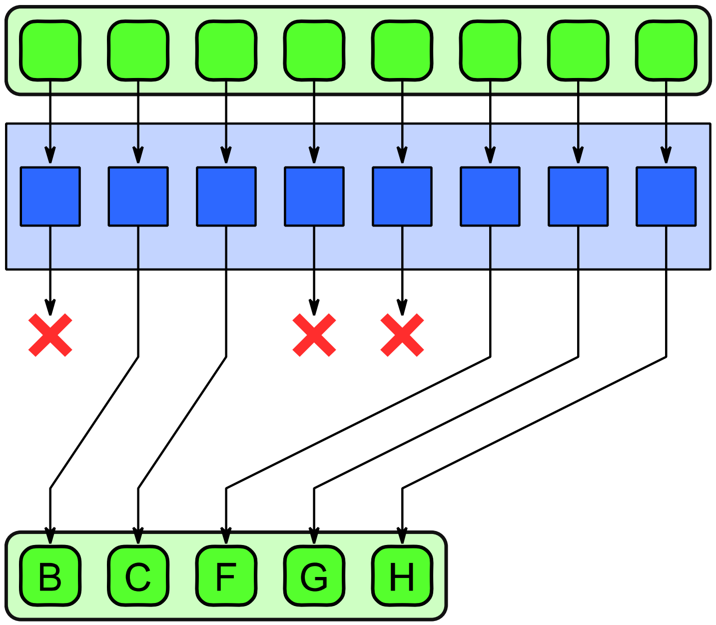
Generalization of Pack: Bin



4 different categories = 4 bins

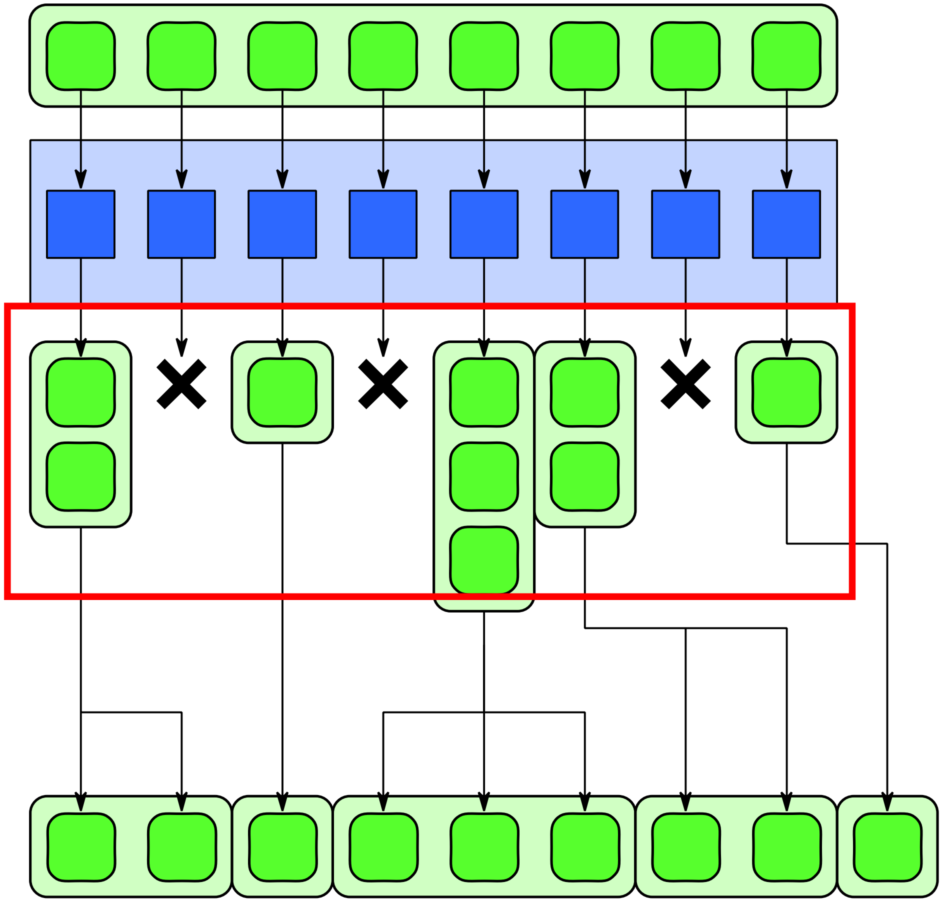
- Generalized split to support more categories (>2)
- Examples
  - Radix sort
  - Pattern classification

Fusion of Map and Pack



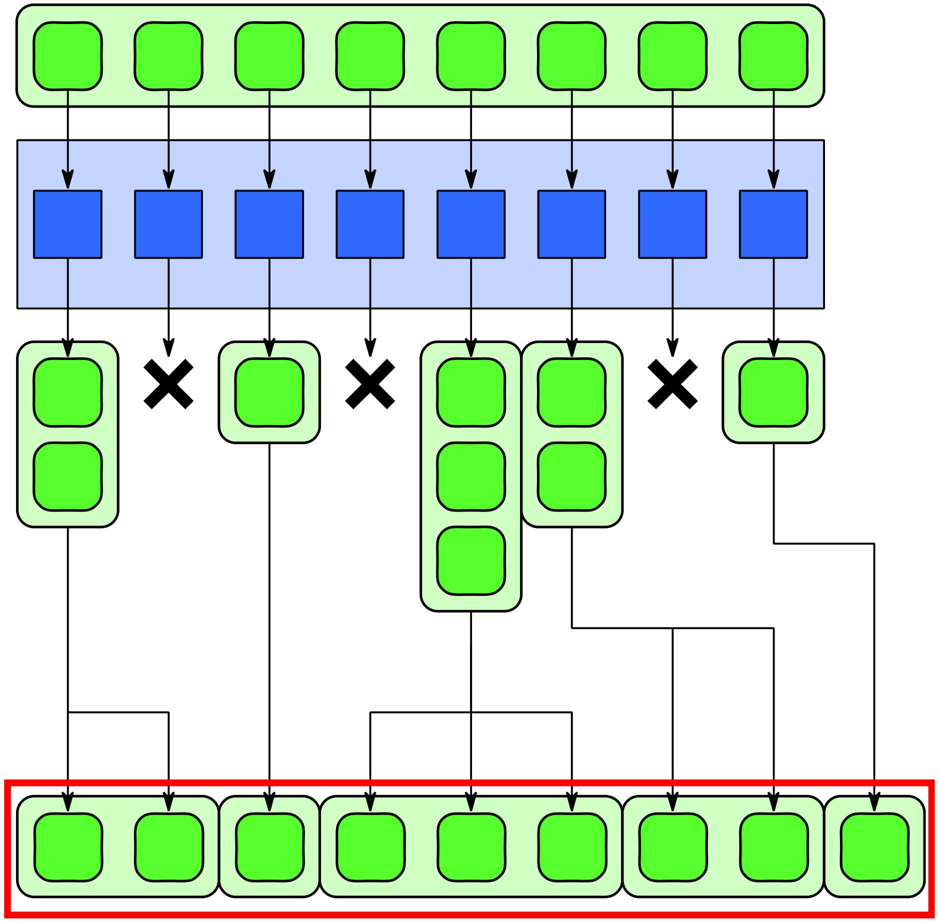
- Advantageous if most of the elements of a map are discarded
- **Map** checks pairs for collision
- **Pack** stores only actual collisions
- Output bandwidth ~ results reported, not number of pairs tested
- Each element can output 0 or 1 element

Generalization of Pack: Expand



- **Each element can output any number of elements**

Generalization of Pack: Expand



- Each element can output any number of elements
- **Results are fused together in order**