058165 - PARALLEL COMPUTING

Fabrizio Ferrandi

a.a. 2022-2023

References

- “Structured Parallel Programming: Patterns for Efficient Computation,” Michael McCool, Arch Robinson, James Reinders, 1st edition, Morgan Kaufmann, ISBN: 978-0-12-415993-8, 2012

Outline

- What are Collectives?
- Reduce Pattern
- Scan Pattern
- Sorting

Collectives

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

Collectives

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

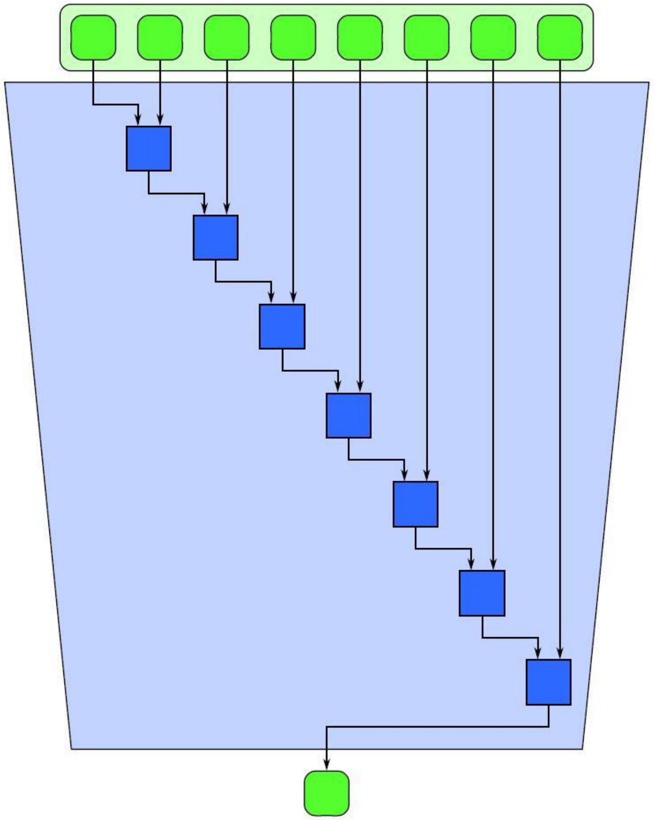
Reduce and Scan will be covered in this lecture

Reduce

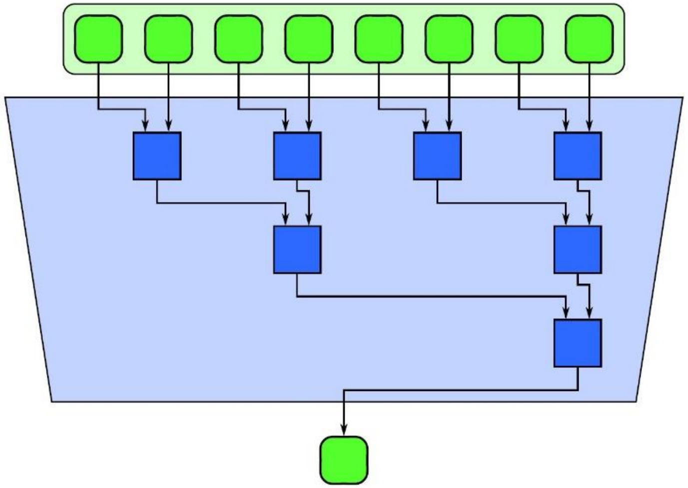
- **Reduce** is used to combine a collection of elements into one summary value
- A combiner function combines elements pairwise
- A combiner function only needs to be *associative* to be parallelizable
- Example combiner functions:
  - Addition
  - Multiplication
  - Maximum / Minimum

Reduce

Serial Reduction

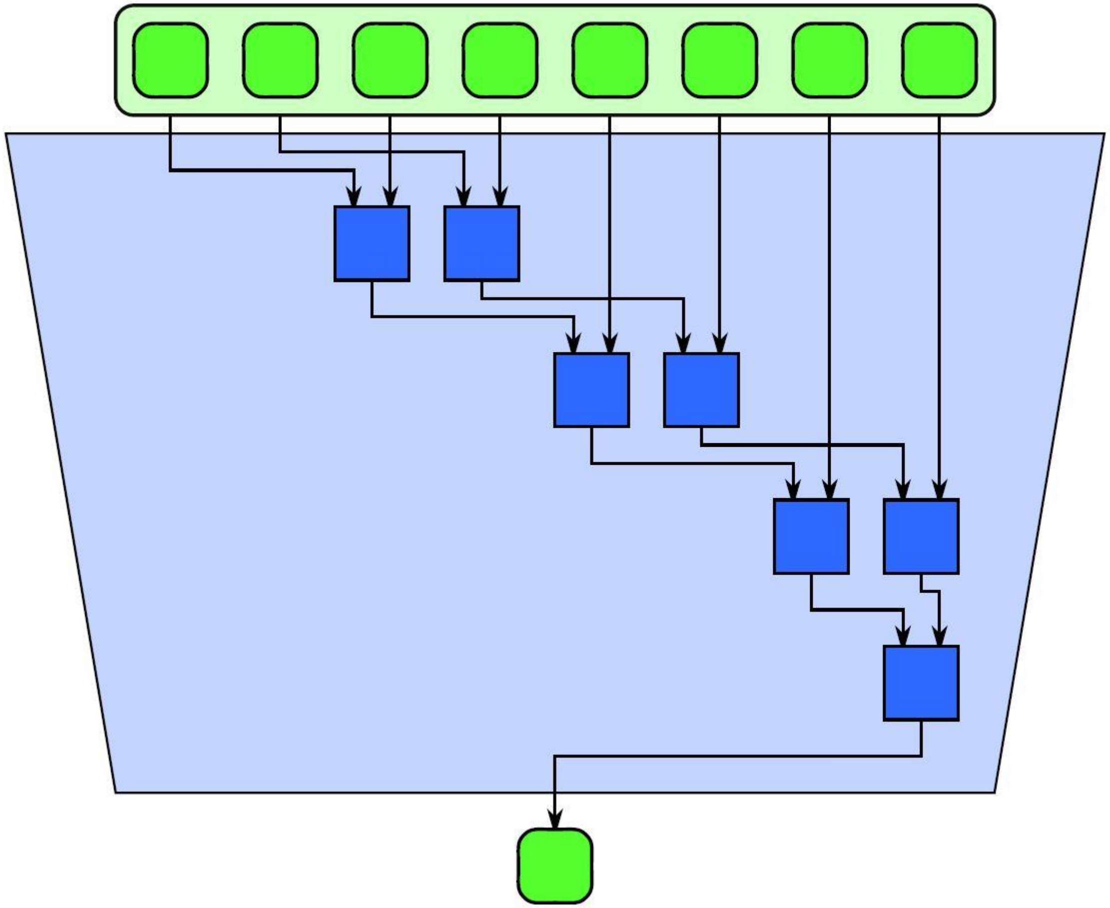


Parallel Reduction



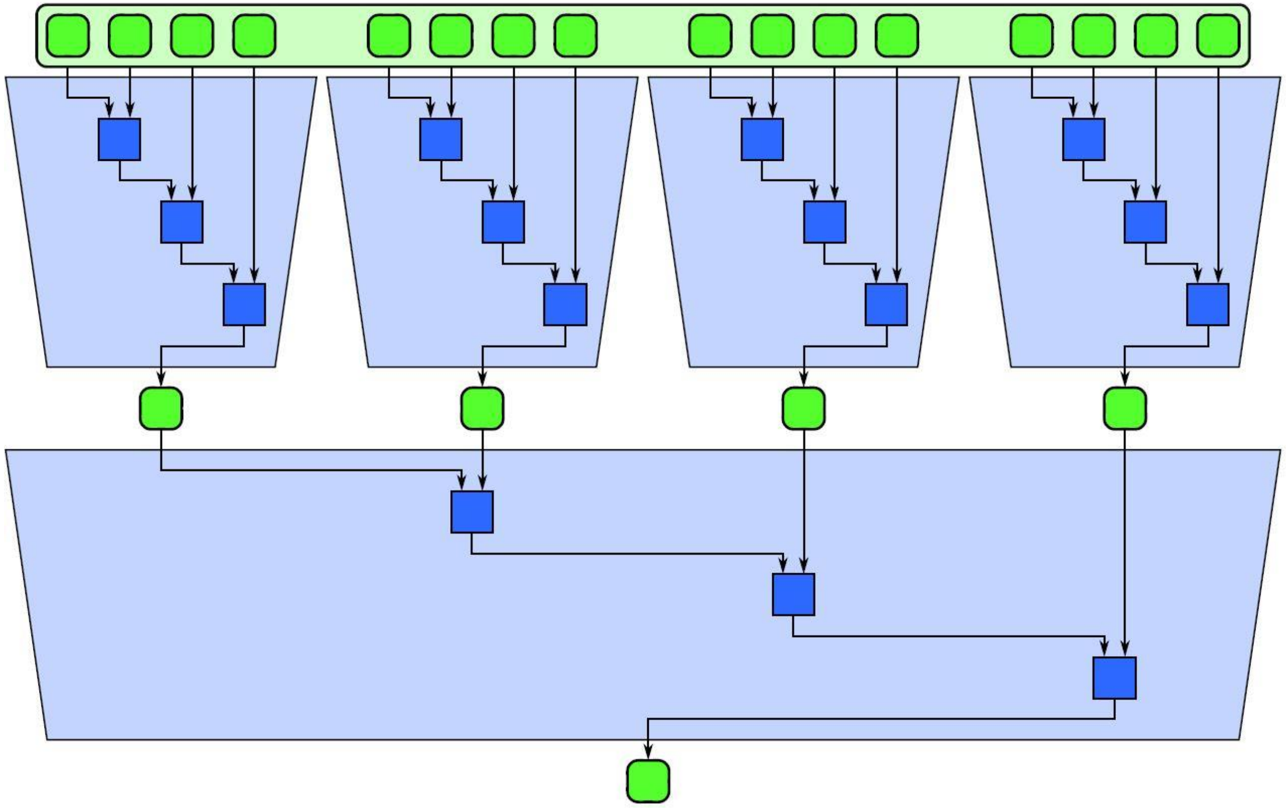
Reduce

- Vectorization

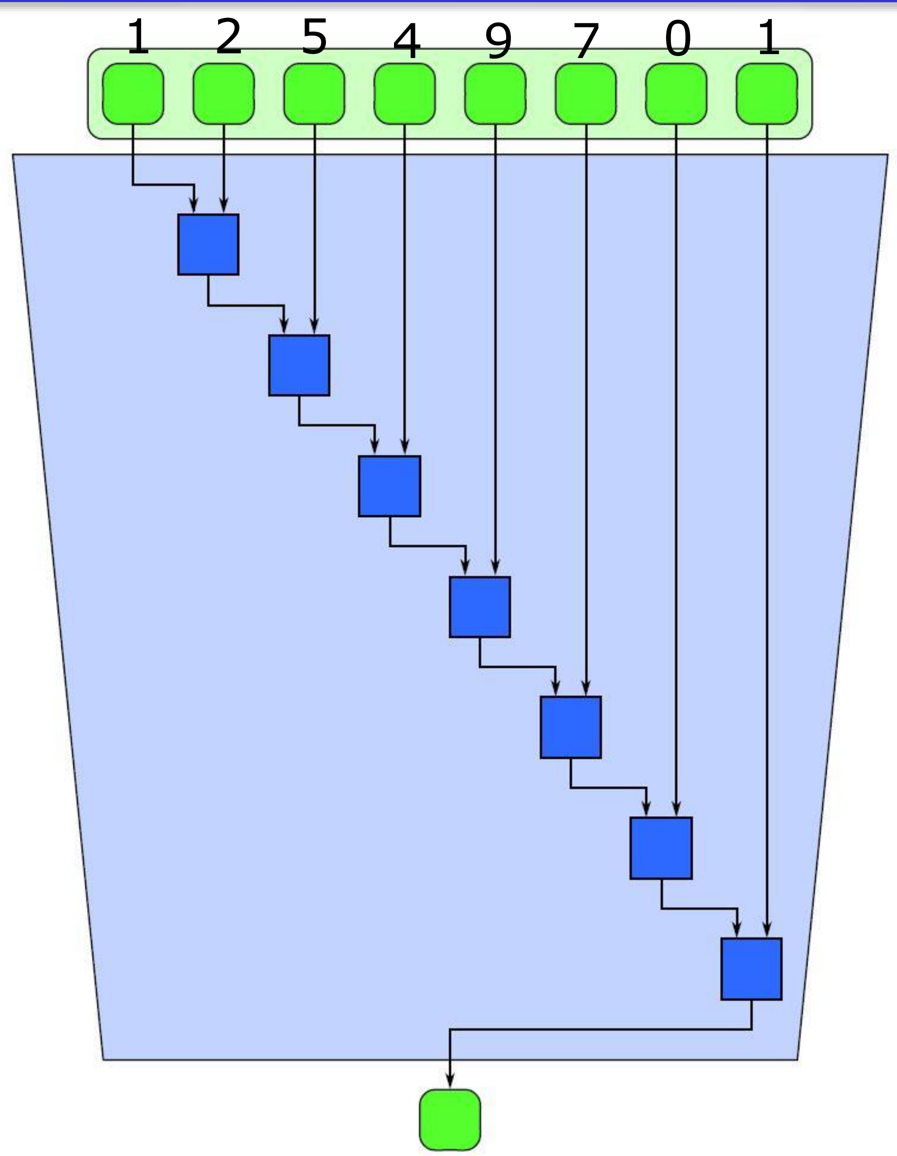


Reduce

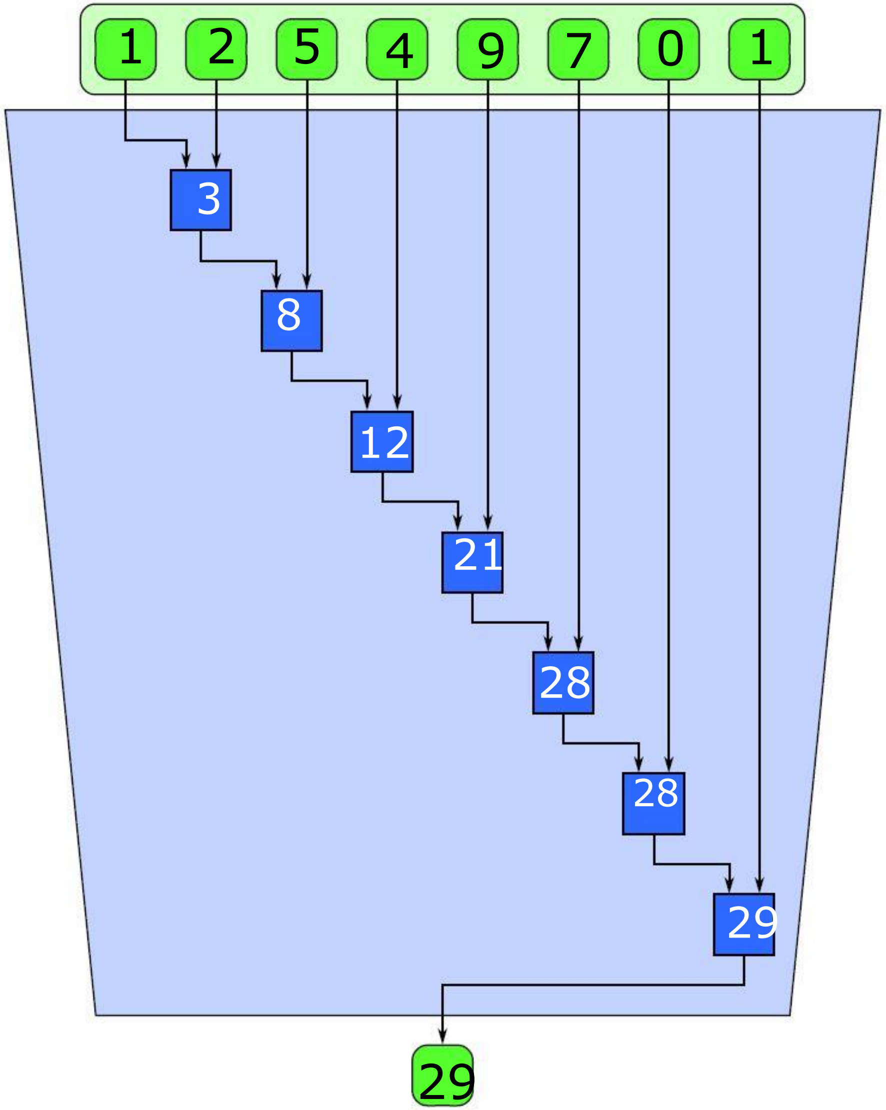
- **Tiling** is used to break chunks of work up for workers to reduce serially



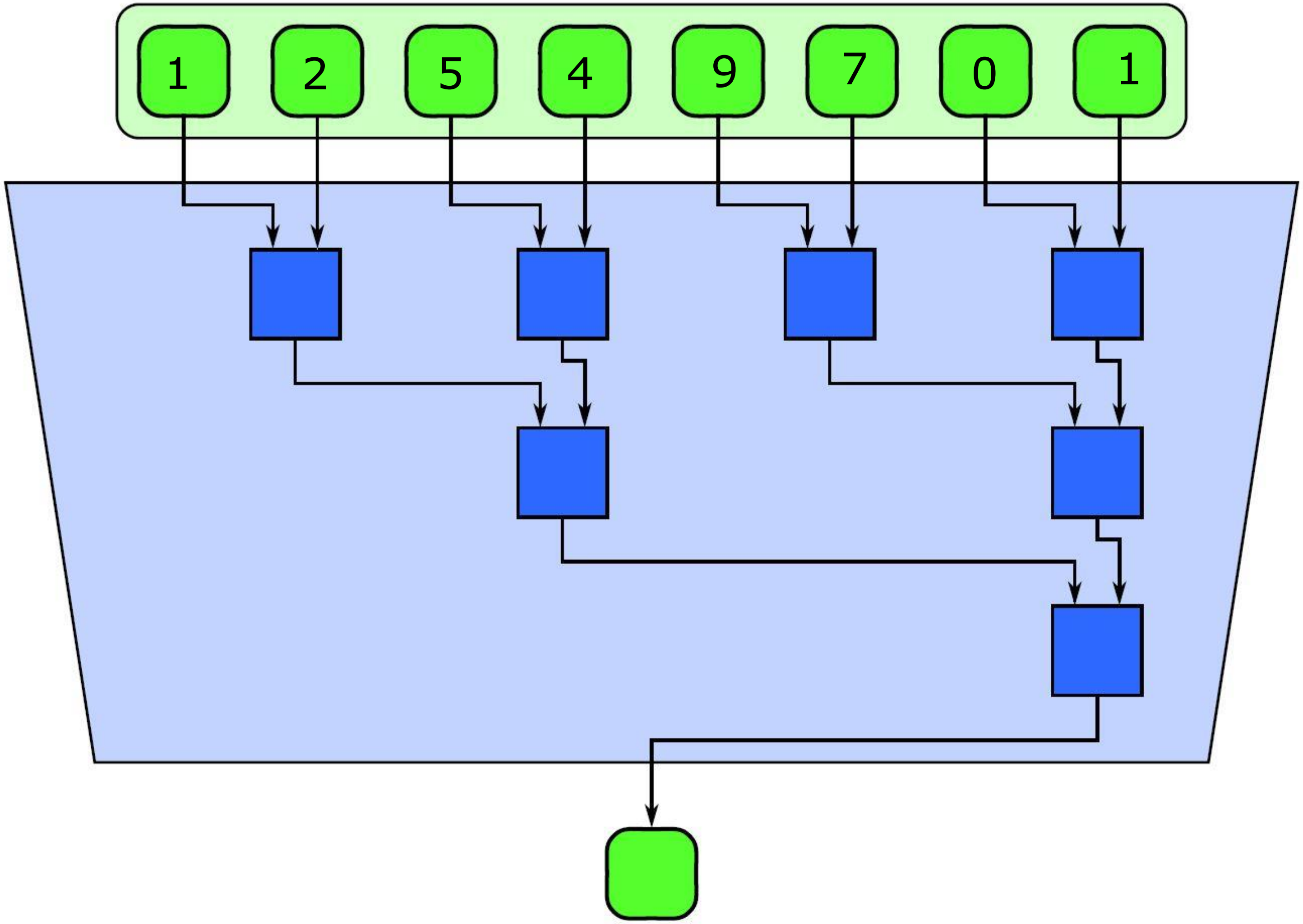
Reduce – Add Example



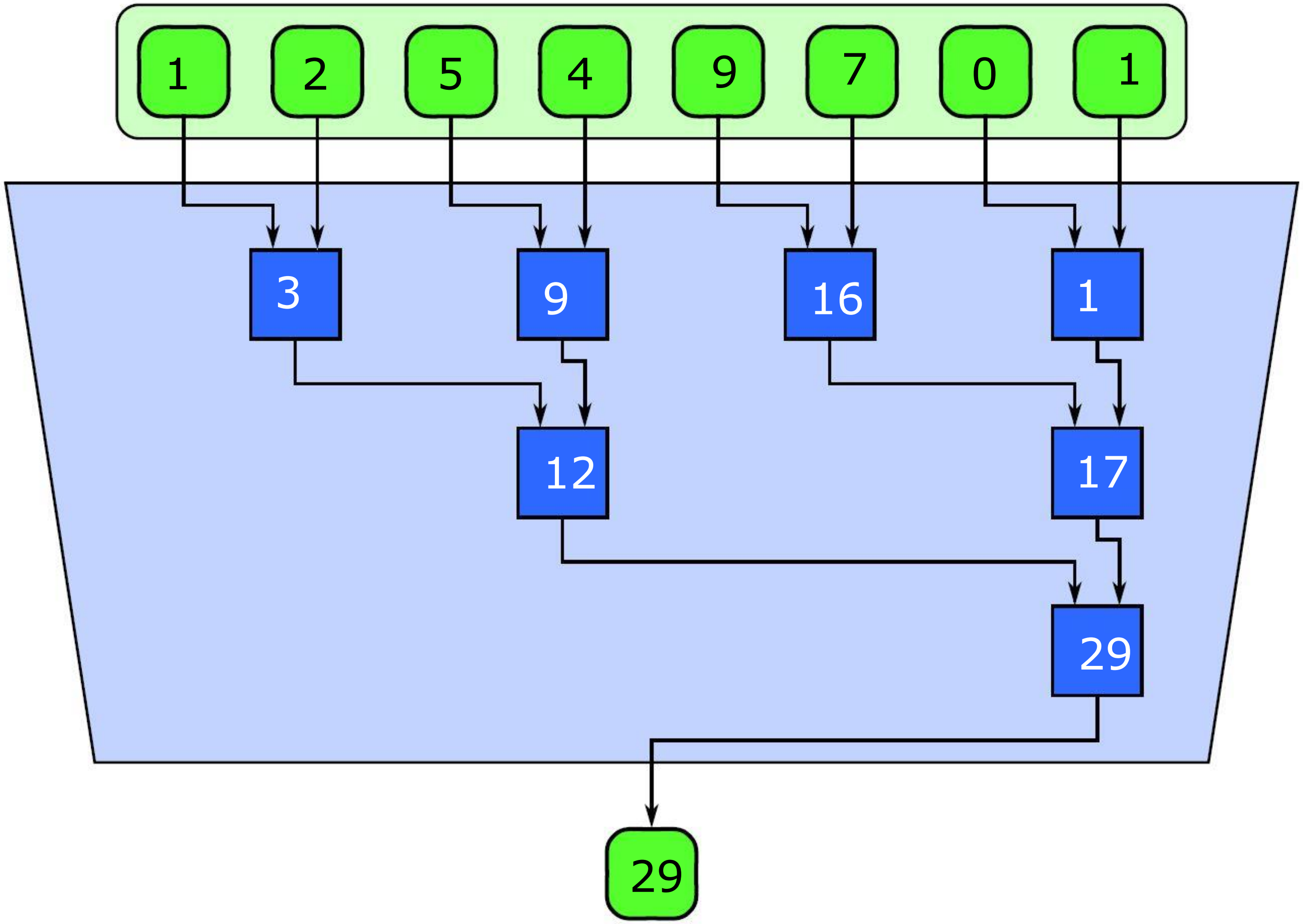
Reduce – Add Example



Reduce – Add Example

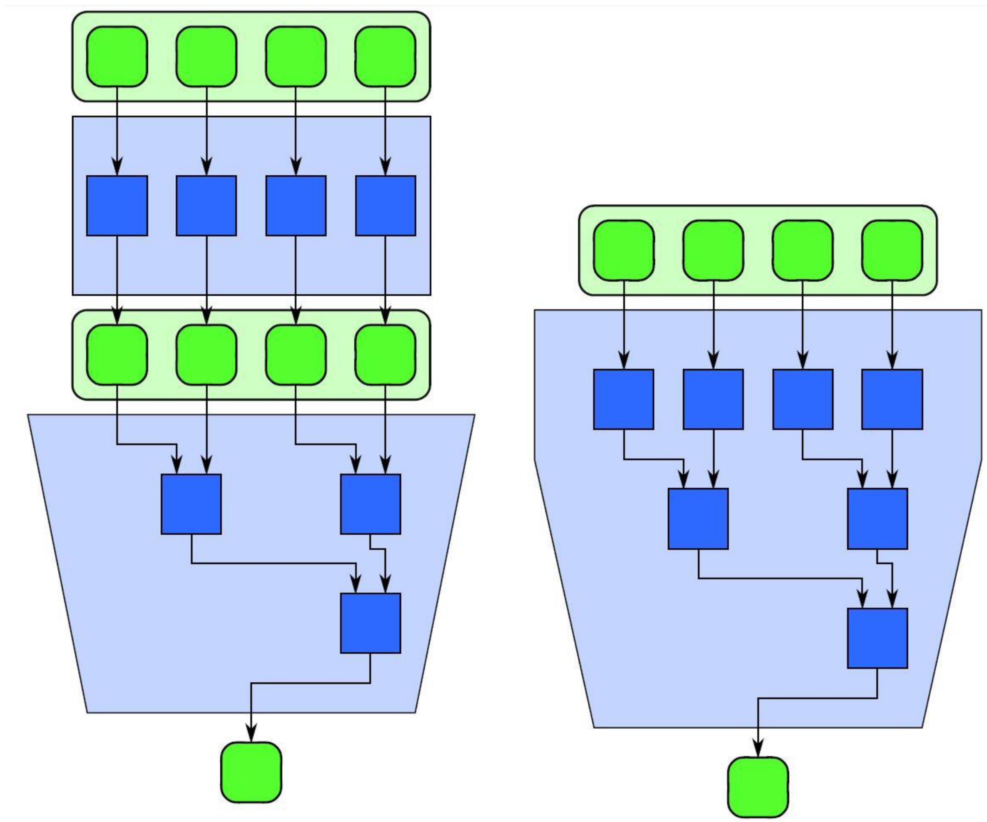


Reduce – Add Example



Reduce

- We can “fuse” the map and reduce patterns



Reduce

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

Reduce Example: Dot Product

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

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

Also: $\vec{a} \cdot \vec{b} = |\vec{a}| \cos(\theta) |\vec{b}|$

Dot Product – Example Uses

- Essential operation in physics, graphics, video games,…
- Gaming analogy: in Mario Kart, there are “boost pads” on the ground that increase your speed
  - red vector is your speed (x and y direction)
  - blue vector is the orientation of the boost pad (x and y direction). Larger numbers are more power.

How much boost will you get? For the analogy, imagine the pad multiplies your speed:

- If you come in going 0, you’ll get nothing
- If you cross the pad perpendicularly, you’ll get 0 [just like the banana obliteration, it will give you 0x boost in the perpendicular direction]

$$Total = speed_x \cdot boost_x + speed_y \cdot boost_y$$

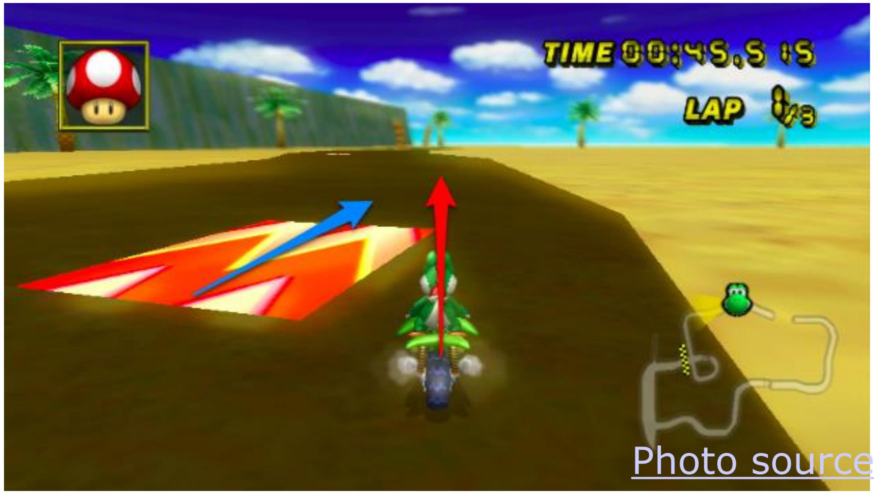


Photo source

Ref: http://betterexplained.com/articles/vector-calculus-understanding-the-dot-product/

Scan

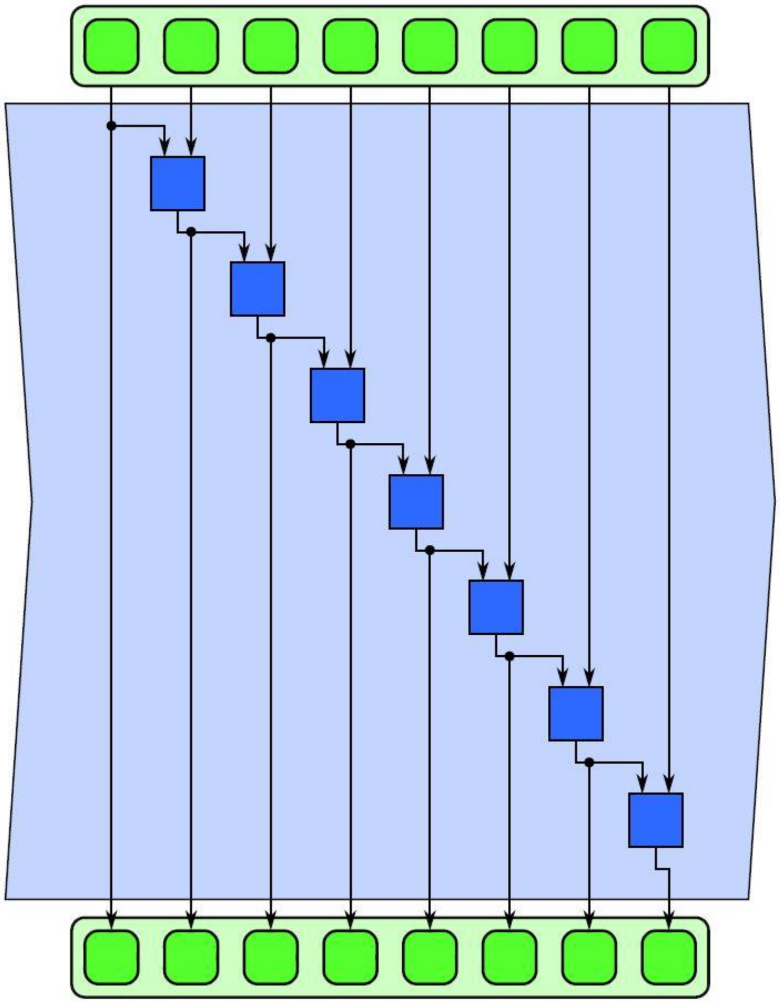
- The **scan** pattern produces partial reductions of input sequence, generates new sequence
- Trickier to parallelize than reduce
- Inclusive scan vs. exclusive scan
  - Inclusive scan: includes current element in partial reduction
  - Exclusive scan: excludes current element in partial reduction, partial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

Scan – Example Uses

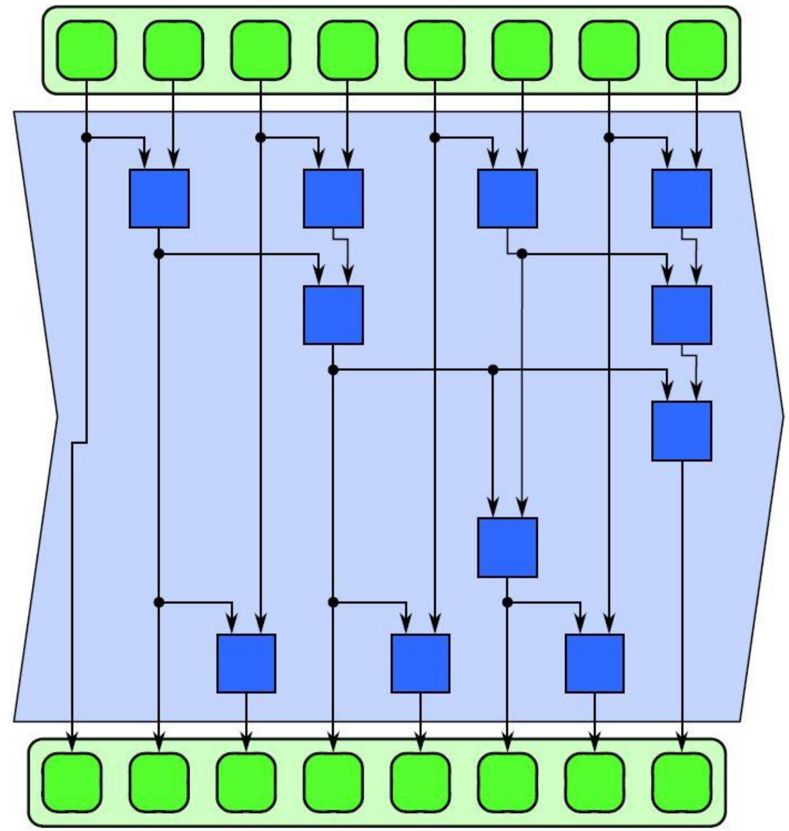
- Lexical comparison of strings – e.g., determine that "strategy" should appear before "stratification" in a dictionary
- Add multi-precision numbers (those that cannot be represented in a single machine *word*)
- Evaluate polynomials
- Implement radix sort or quicksort
- Delete marked elements in an 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

Scan

Serial Scan

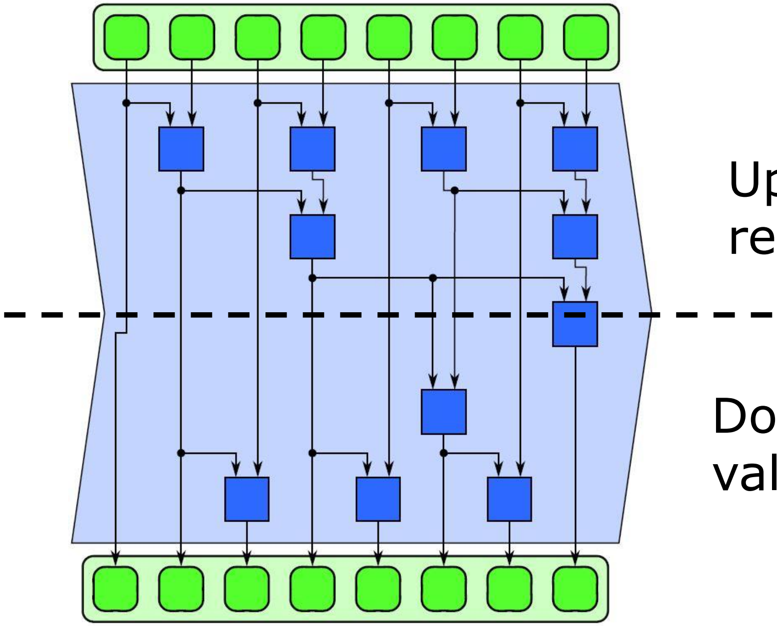


Parallel Scan



Scan

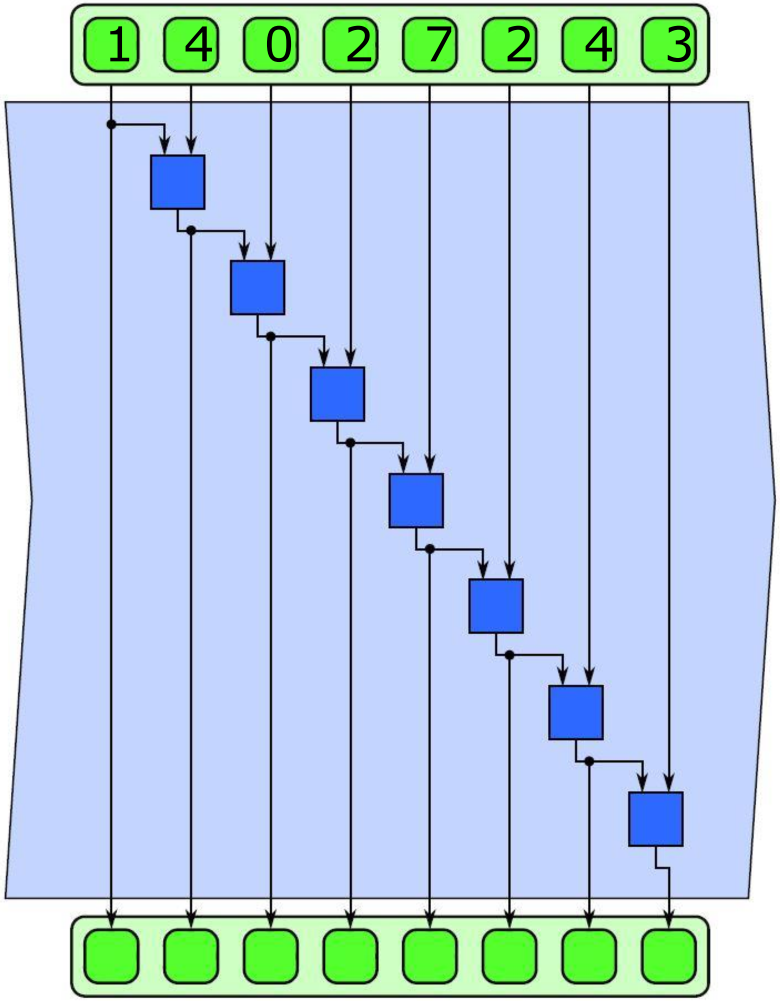
- One algorithm for parallelizing scan is to perform an "up sweep" and a "down sweep"
- Reduce the input on the up sweep
- The down sweep produces the intermediate results

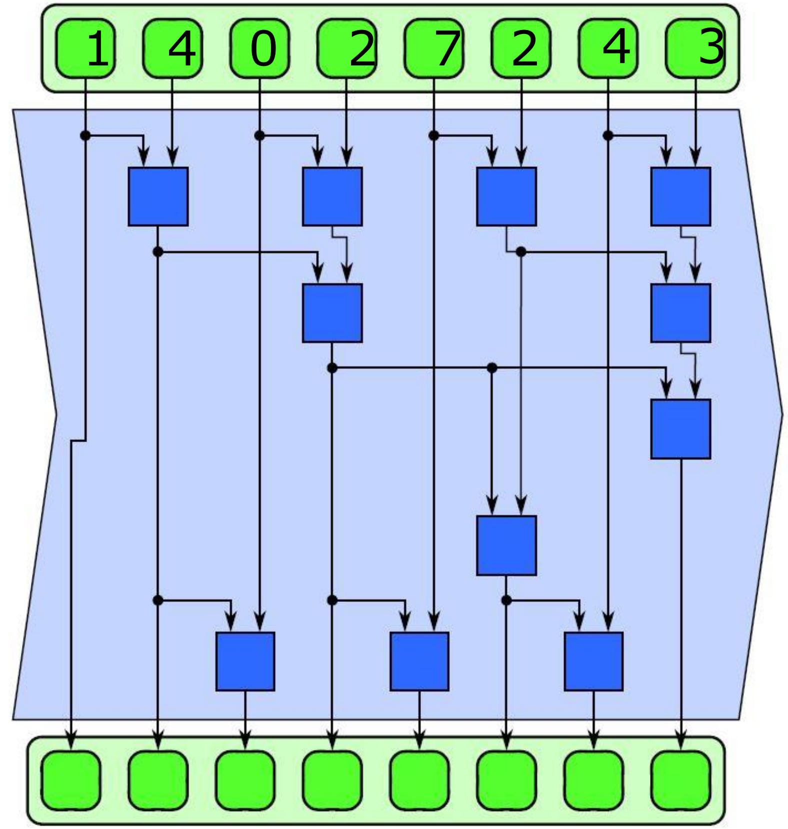


Up sweep – compute reduction

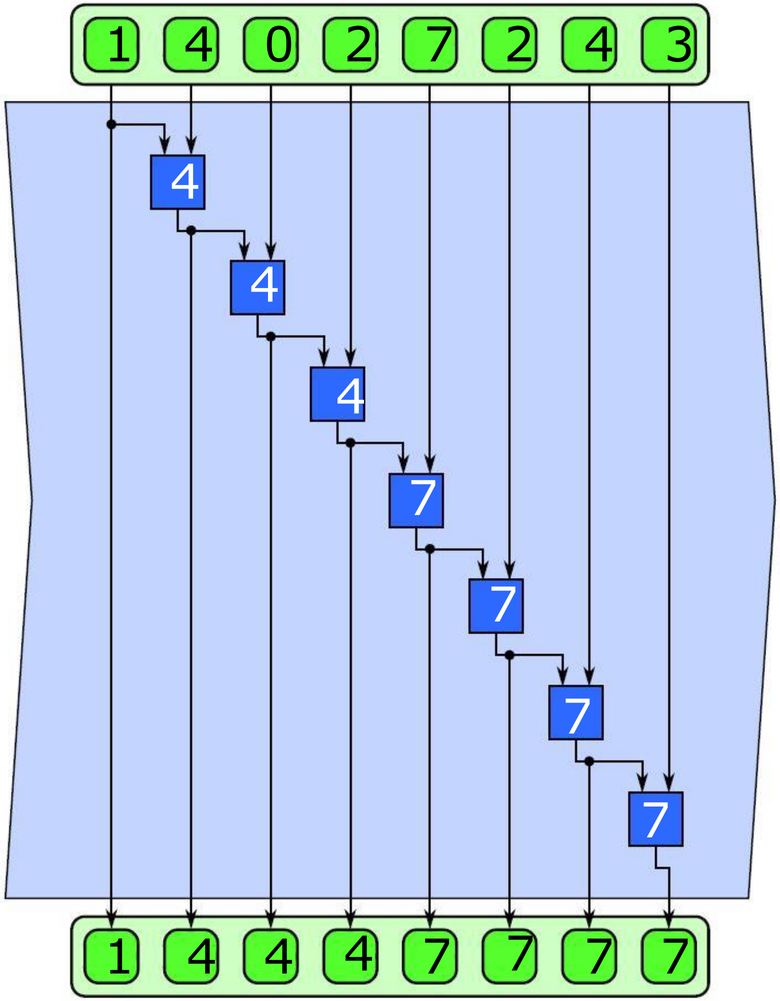
Down sweep – compute intermediate values

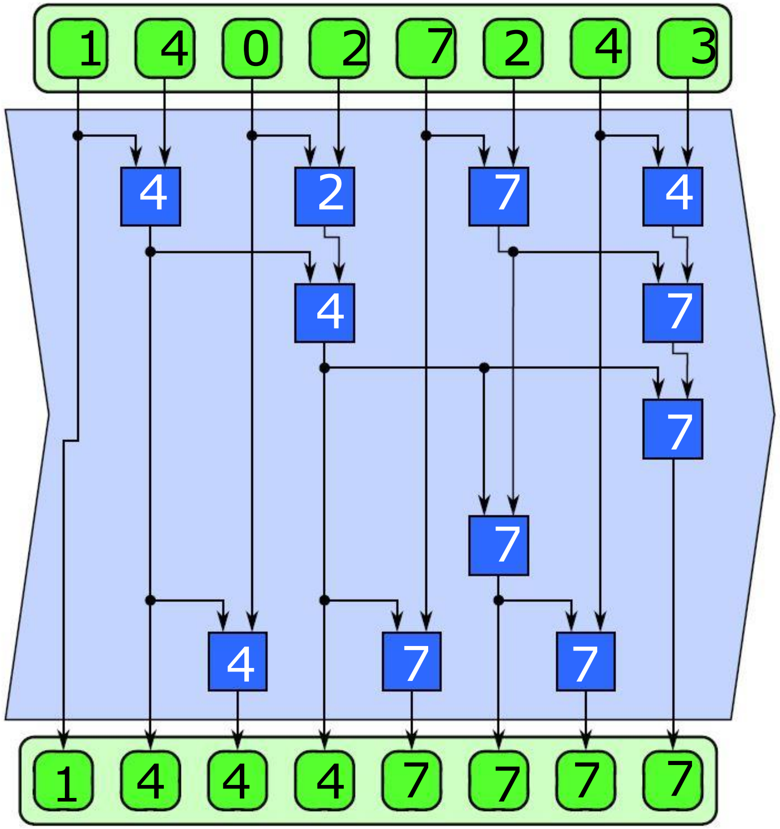
Scan – Maximum Example





Scan – Maximum Example

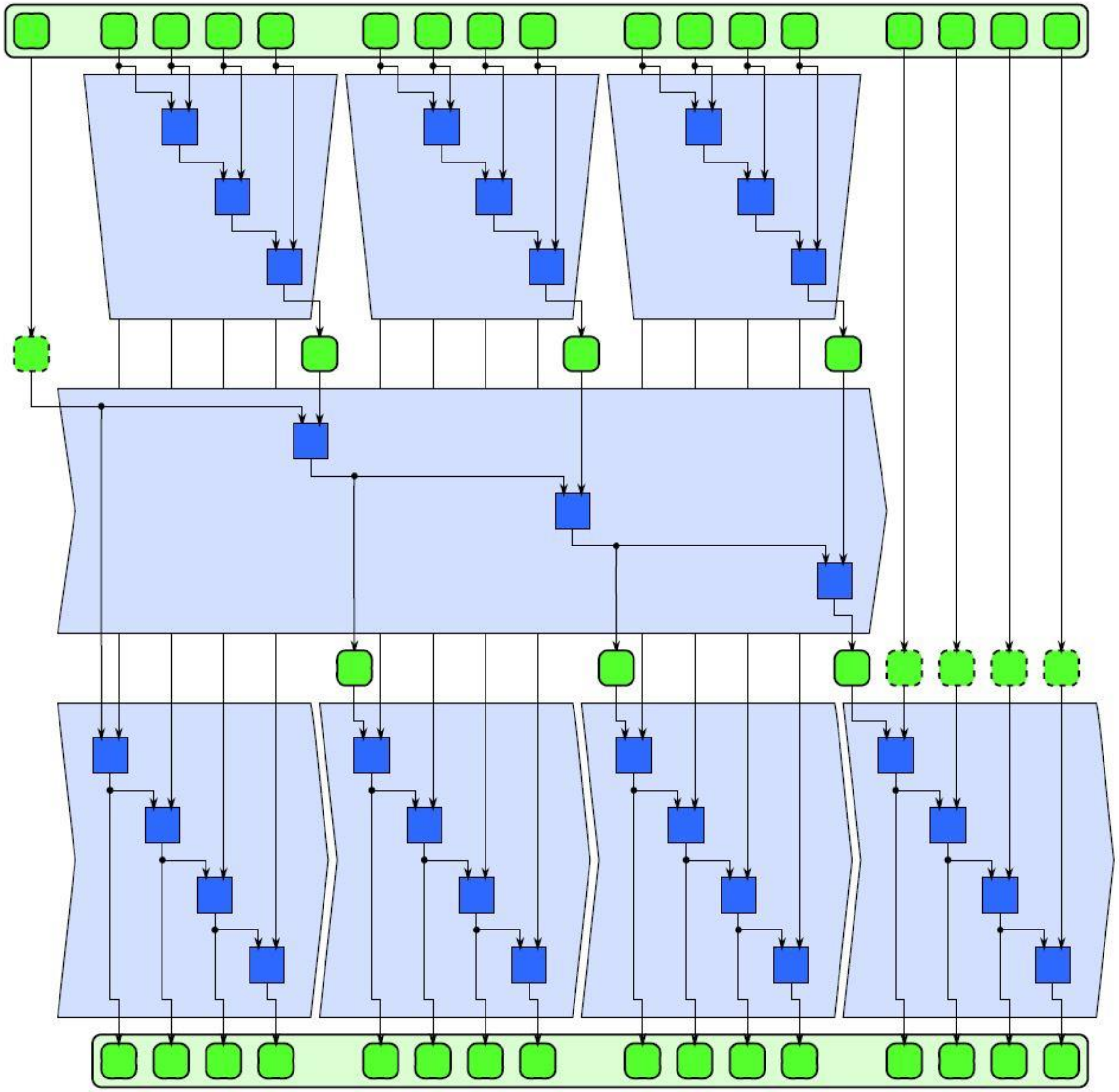




Scan

- Three phase scan with tiling

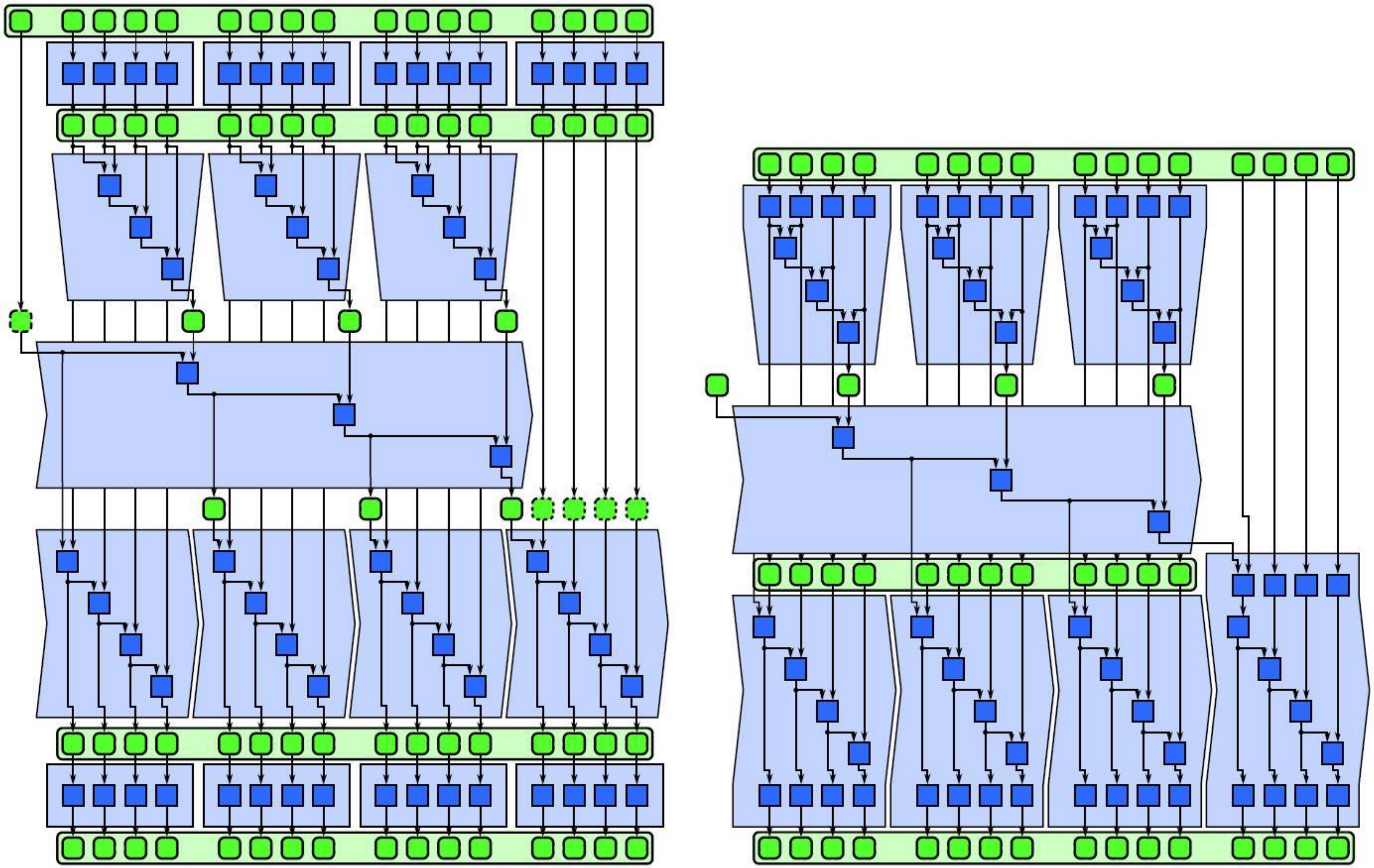
Scan



Scan

- Just like reduce, we can also fuse the **map** pattern with the **scan** pattern

Scan



Merge Sort as a reduction

- We can sort an array via a pair of a map and a reduce
- Map each element into a vector containing just that element
  - <> is the merge operation: [1,3,5,7] <> [2,6,15] = [1,2,3,5,6,7,15]
  - [] is the empty list
- How fast is this?

Right Biased Sort

Start with [14,3,4,8,7,52,1]

Map to [[14],[3],[4],[8],[7],[52],[1]]

Reduce:

[14] <> ([3] <> ([4] <> ([8] <> ([7] <> ([52] <> [1]))))))

= [14] <> ([3] <> ([4] <> ([8] <> ([7] <> [1,52]))))

= [14] <> ([3] <> ([4] <> ([8] <> [1,7,52])))

= [14] <> ([3] <> ([4] <> [1,7,8,52]))

= [14] <> ([3] <> [1,4,7,8,52])

= [14] <> [1,3,4,7,8,52]

= [1,3,4,7,8,14,52]

Right Biased Sort Cont

- How long did that take?
- We did O(n) merges…but each one took O(n) time
- $O(n^2)$
- We wanted merge sort, but instead we got insertion sort!

Tree Shape Sort

Start with [14,3,4,8,7,52,1]

Map to [[14],[3],[4],[8],[7],[52],[1]]

Reduce:

(([14] <> [3]) <> ([4] <> [8])) <> (([7] <> [52]) <> [1])

= ([3,14] <> [4,8]) <> ([7,52] <> [1])

= [3,4,8,14] <> [1,7,52]

= [1,3,4,7,8,14,52]

Tree Shaped Sort Performance

- Even if we only had a single processor this is better
  - We do O(log n) merges
  - Each one is O(n)
  - So O(n*log(n))
- But opportunity for parallelism is not so great
  - O(n) assuming sequential merge
  - Takeaway: the shape of reduction matters!