NOVA SCHOOL OF SCIENCE & TECHNOLOGY
COMPUTER SCIENCE DEPARTMENT

Map and Reduce Patterns

lecture 09 (2021-04-12)

Master in Computer Science and Engineering

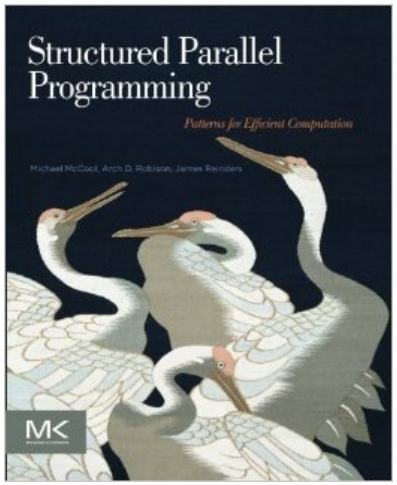
— Concurrency and Parallelism / 2020-21 —

João Lourenço <joao.lourenco@fct.unl.pt>

Outline

- Map pattern
  - Optimizations
    - sequences of Maps
    - code Fusion
    - cache Fusion
  - Related Patterns
  - Example: Scaled Vector Addition (SAXPY)
- Reduce
  - Example: Dot Product

- Bibliography:
  - **Chapters 4 and 5** of book McCool M., Arch M., Reinders J.; Structured Parallel Programming: Patterns for Efficient Computation; Morgan Kaufmann (2012); ISBN: 978-0-12-415993-8



Mapping

- "Do the same thing many times"

```
foreach i in foo:

    do_something(i)
```

- Well-known higher order function in languages like ML, Haskell, Scala

$$\texttt{map}: \forall ab.(a \to b) List\langle a \rangle \to List\langle b \rangle$$

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

Independence

- The key to (embarrassing) parallelism is independence

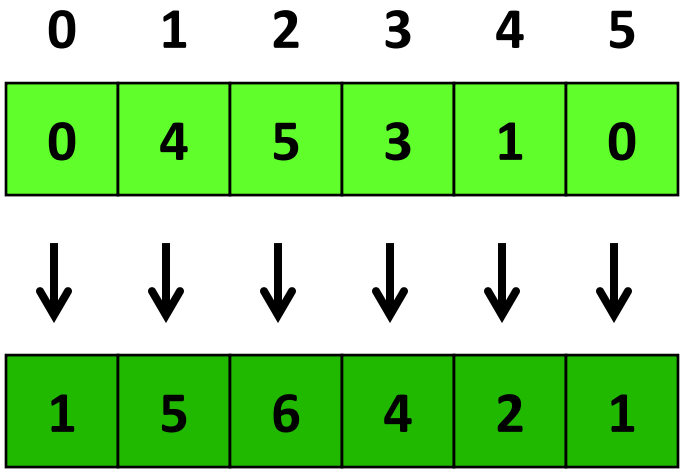
Warning: No shared state!

Map function should be "pure" and should not modify shared states

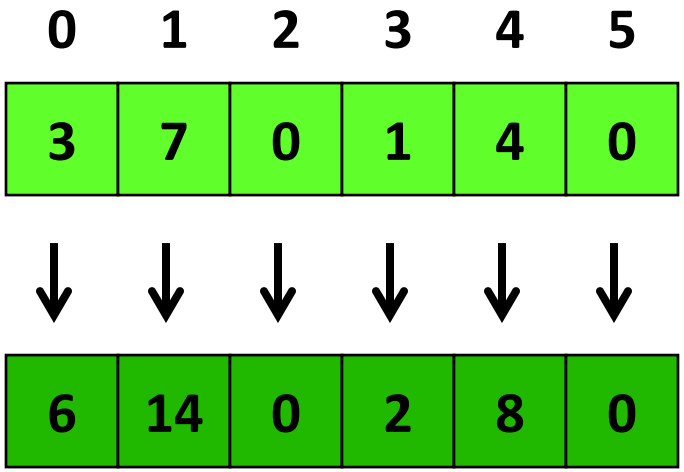
- Modifying shared state breaks perfect independence
- Possible results of accidentally violating independence:
  - non-determinism
  - data-races
  - undefined behavior
  - segfaults

Example Maps

Add 1 to every item in an array



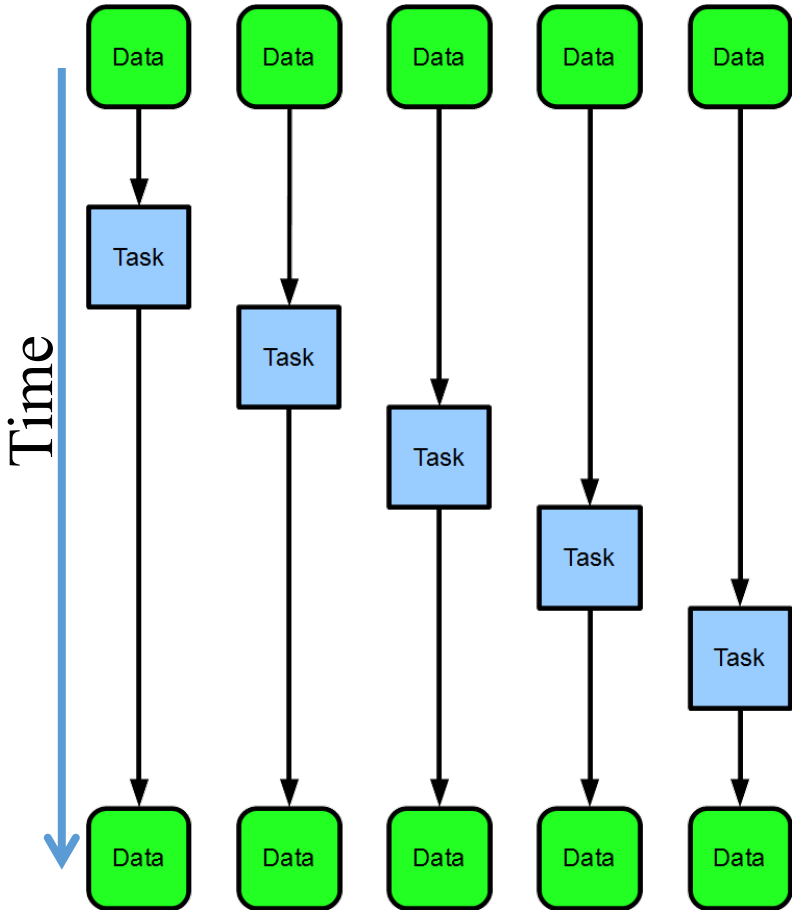
Double every item in an array



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

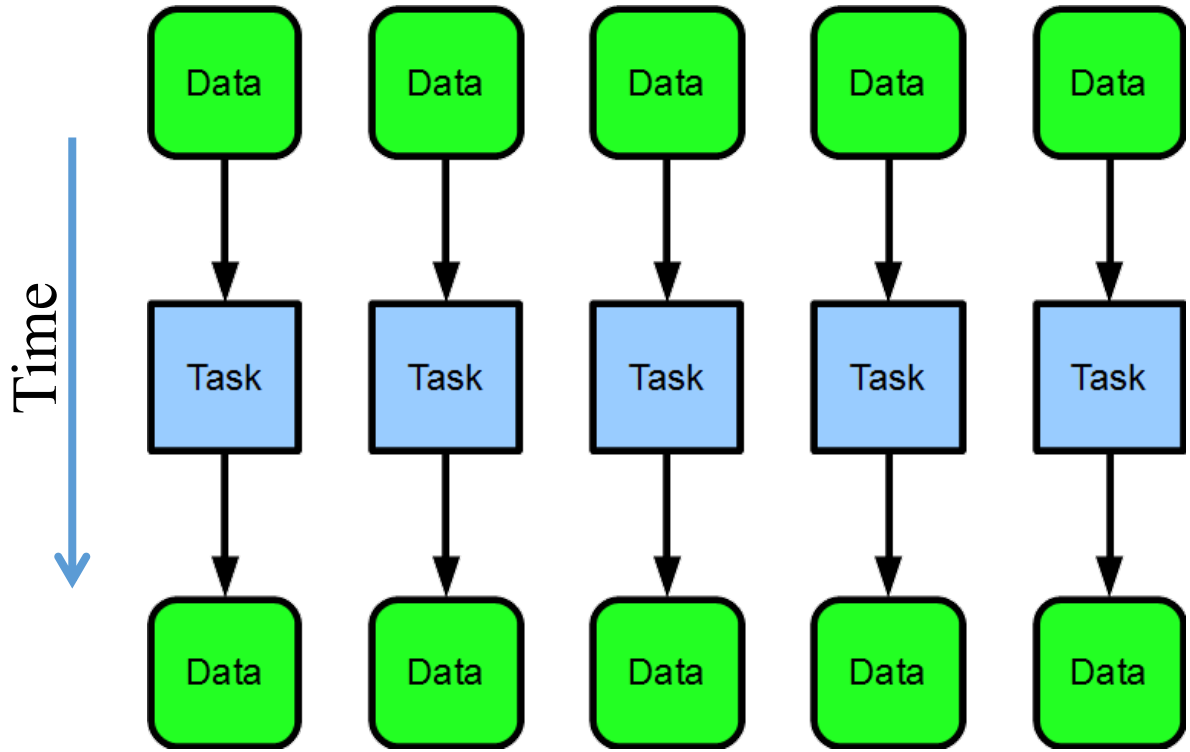
Sequential Map

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

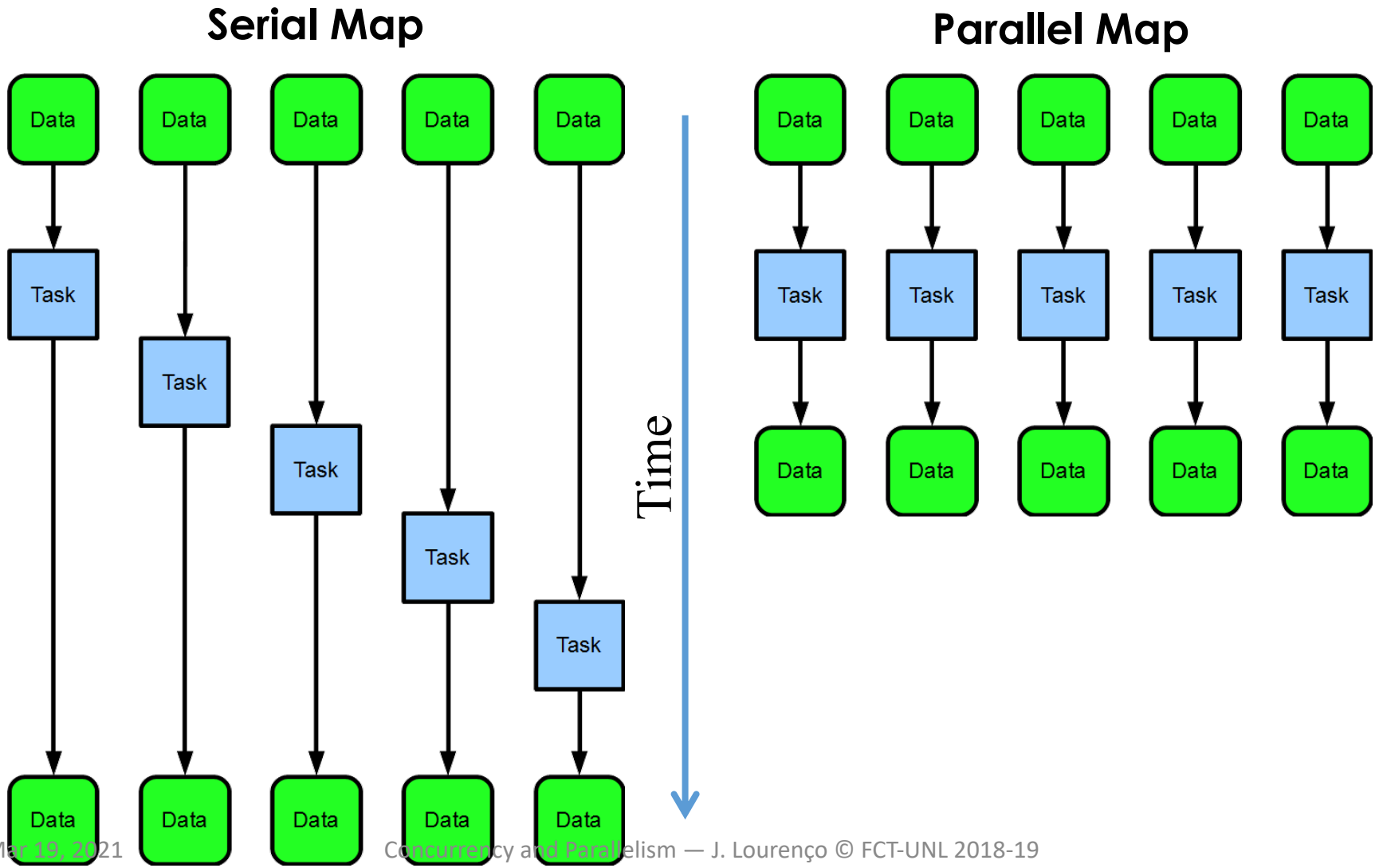


Parallel Map

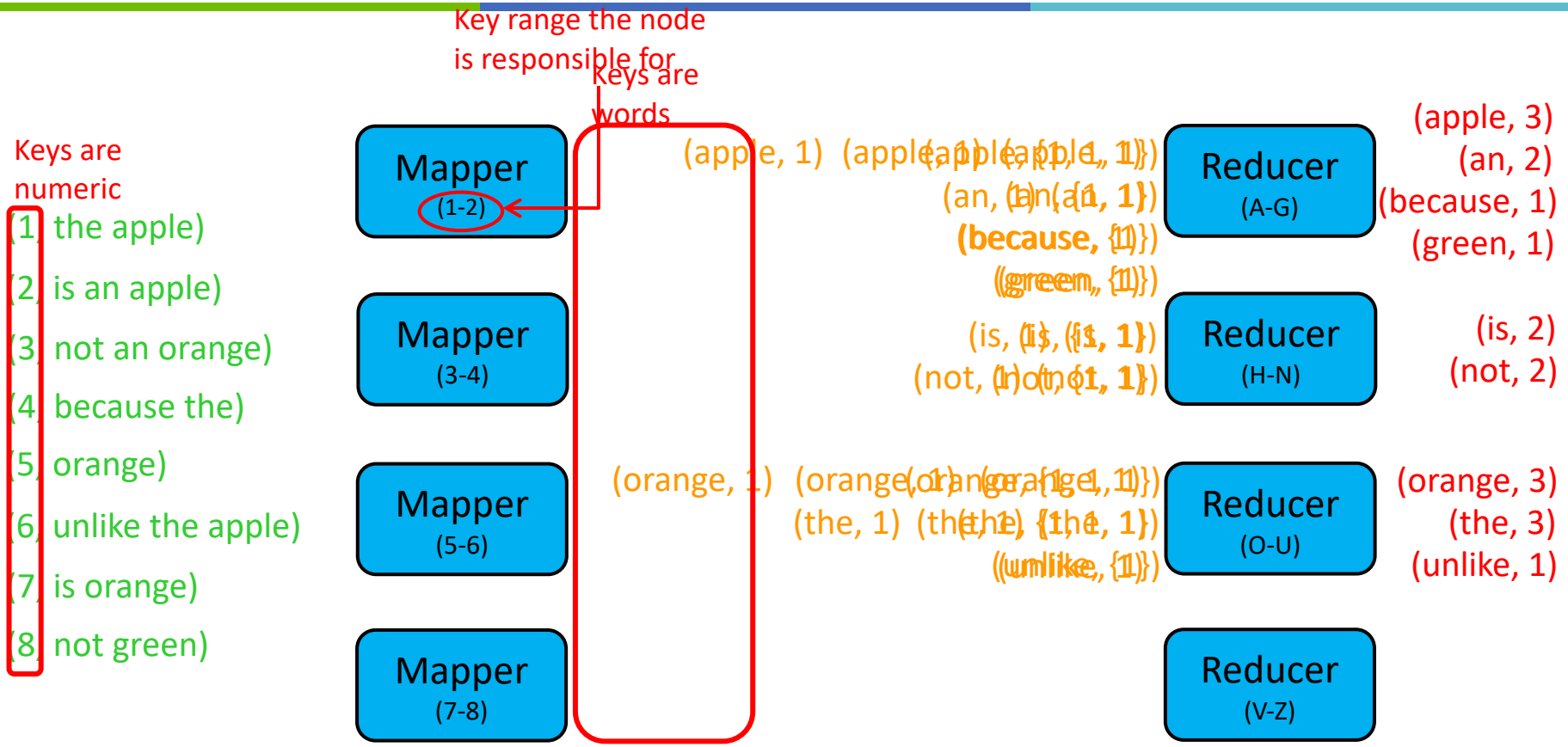
```
parallel_for_each(x in array){
        process(x);
}
```



Comparing Maps



Simple example: Word count



1. Each mapper receives some of the KV-pairs as input
2. The mappers process the KV-pairs one by one
3. Each KV-pair output by the mapper is sent to the reducer that is responsible for it
4. The infrastructure sort their input by key and group it
5. The reducers process their input one group at a time

Implementation and API

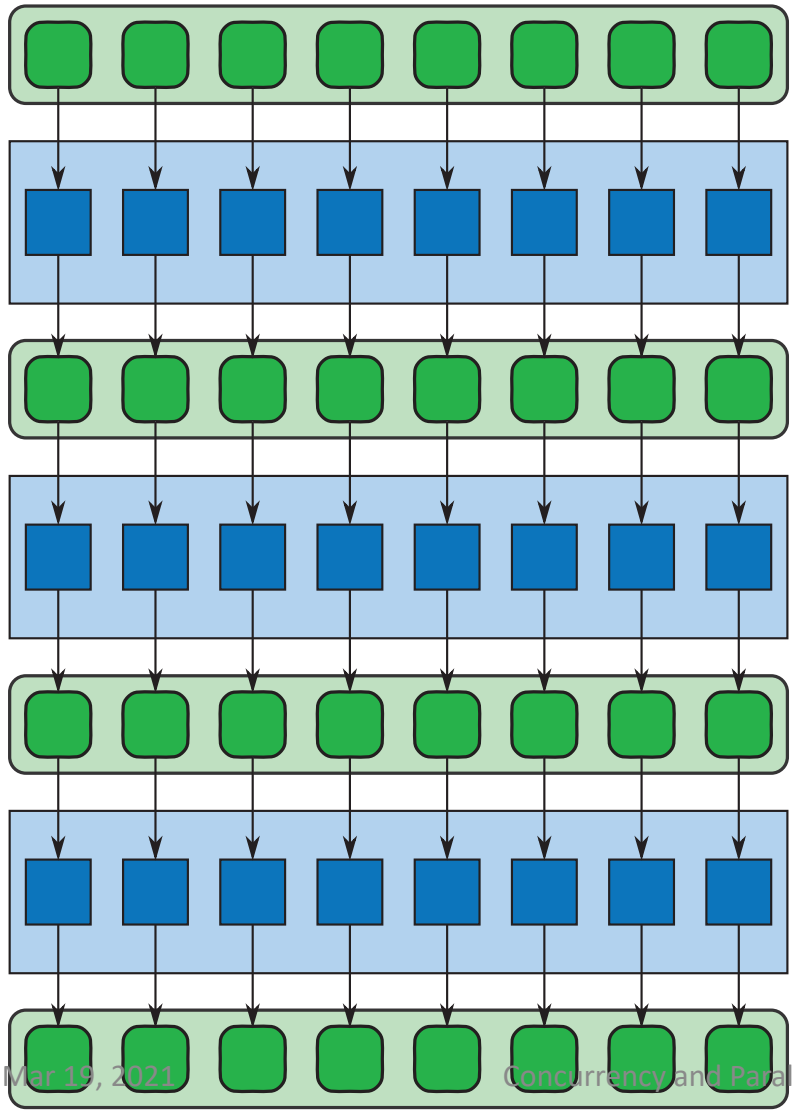
- OpenMP contain a parallel ***for*** language construct
- Map is a mode of use of parallel ***for***
- Some languages (CilkPlus, Matlab, Fortran) provide **array notation** which makes some maps more concise

Array Notation

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

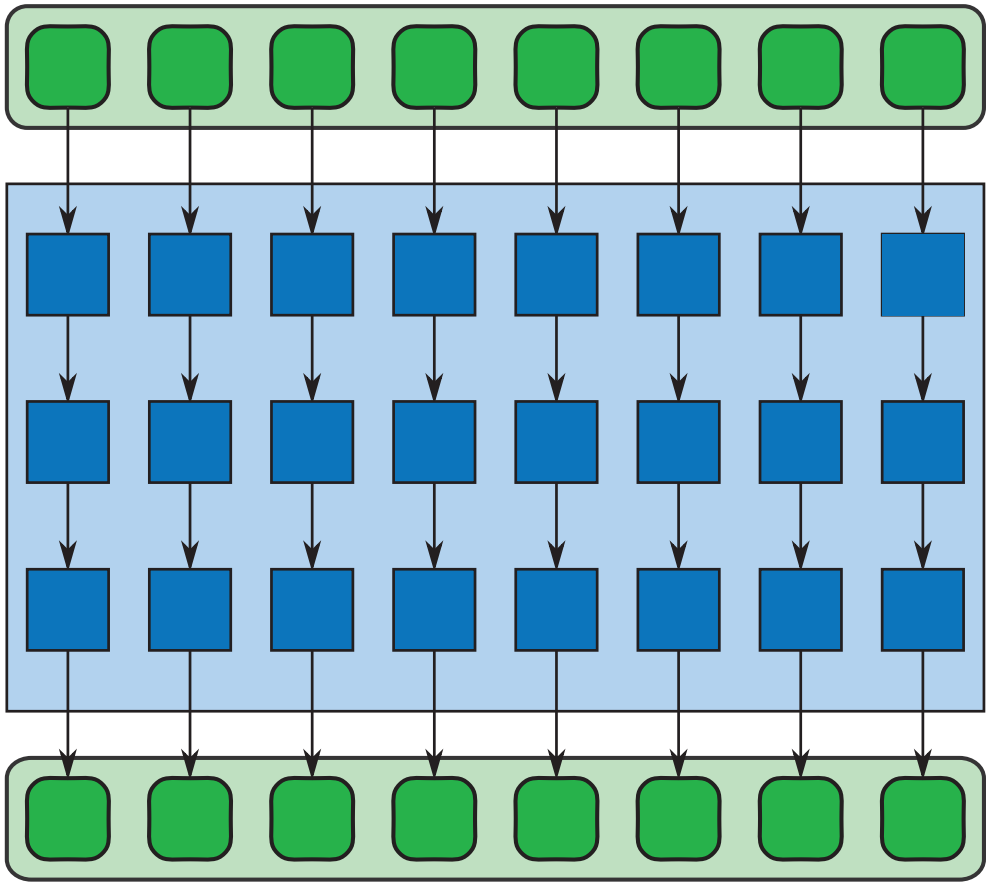
is CilkPlus array notation for "multiply every element in *A* by 5"

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 BW and likely overwhelming the cache

Optimization – Code Fusion



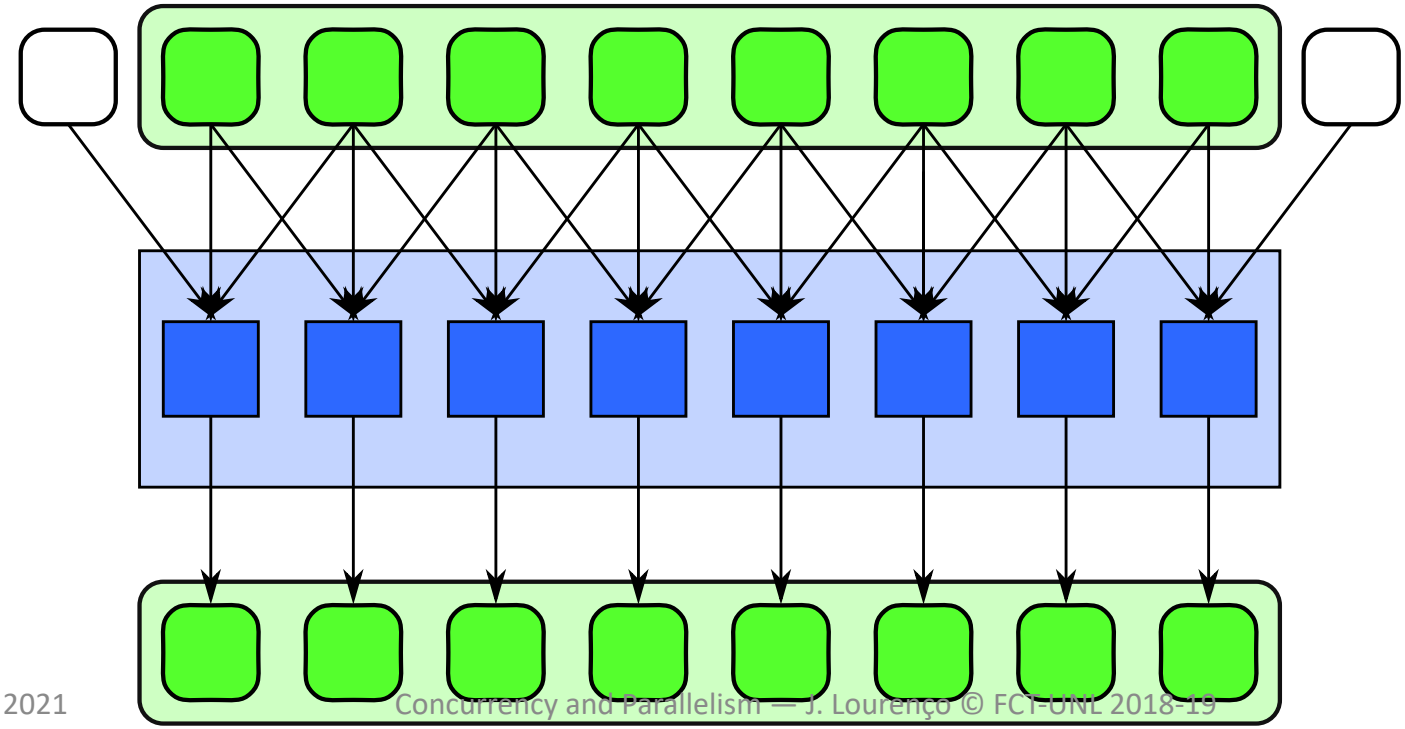
- Can sometimes "fuse" together the operations to perform them at once
- Adds arithmetic intensity, reduces memory/cache usage
- Ideally, operations can be performed using registers alone

Related Patterns

- Three patterns related to map are now discussed here:
  - Stencil
  - Workpile
  - Divide-and-Conquer

Stencil

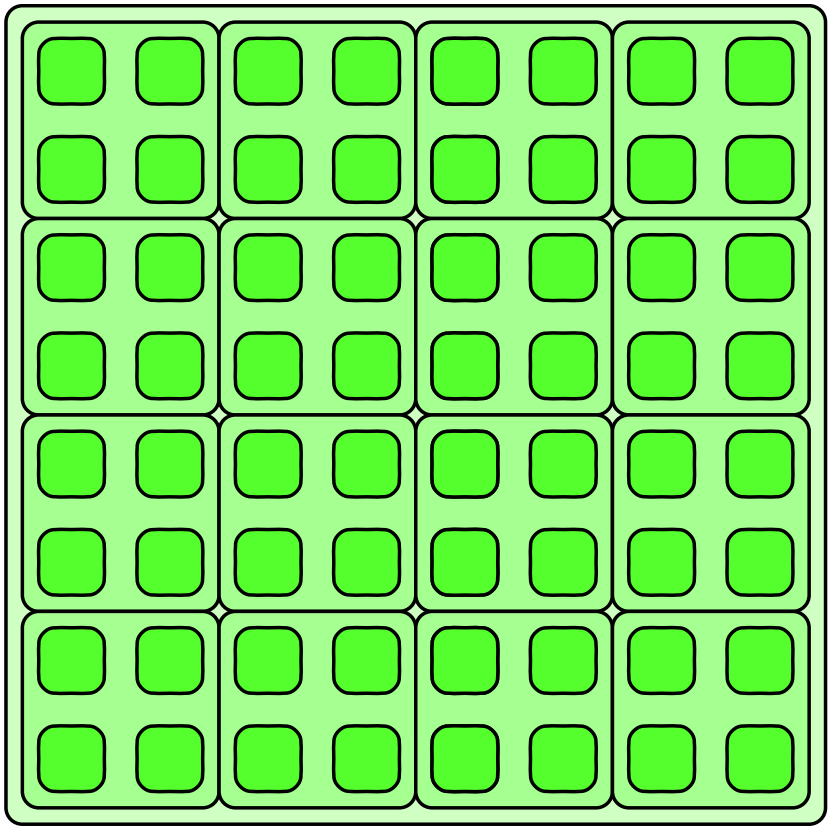
- Each instance of the map function accesses neighbors of its input, offset from its usual input
- Common in imaging and PDE solvers



Workpile (master-slave)

- Work items can be added to the map while it is in progress, from inside map function instances
- Work grows and is consumed by the map
- Workpile pattern terminates when no more work is available

Divide-and-Conquer

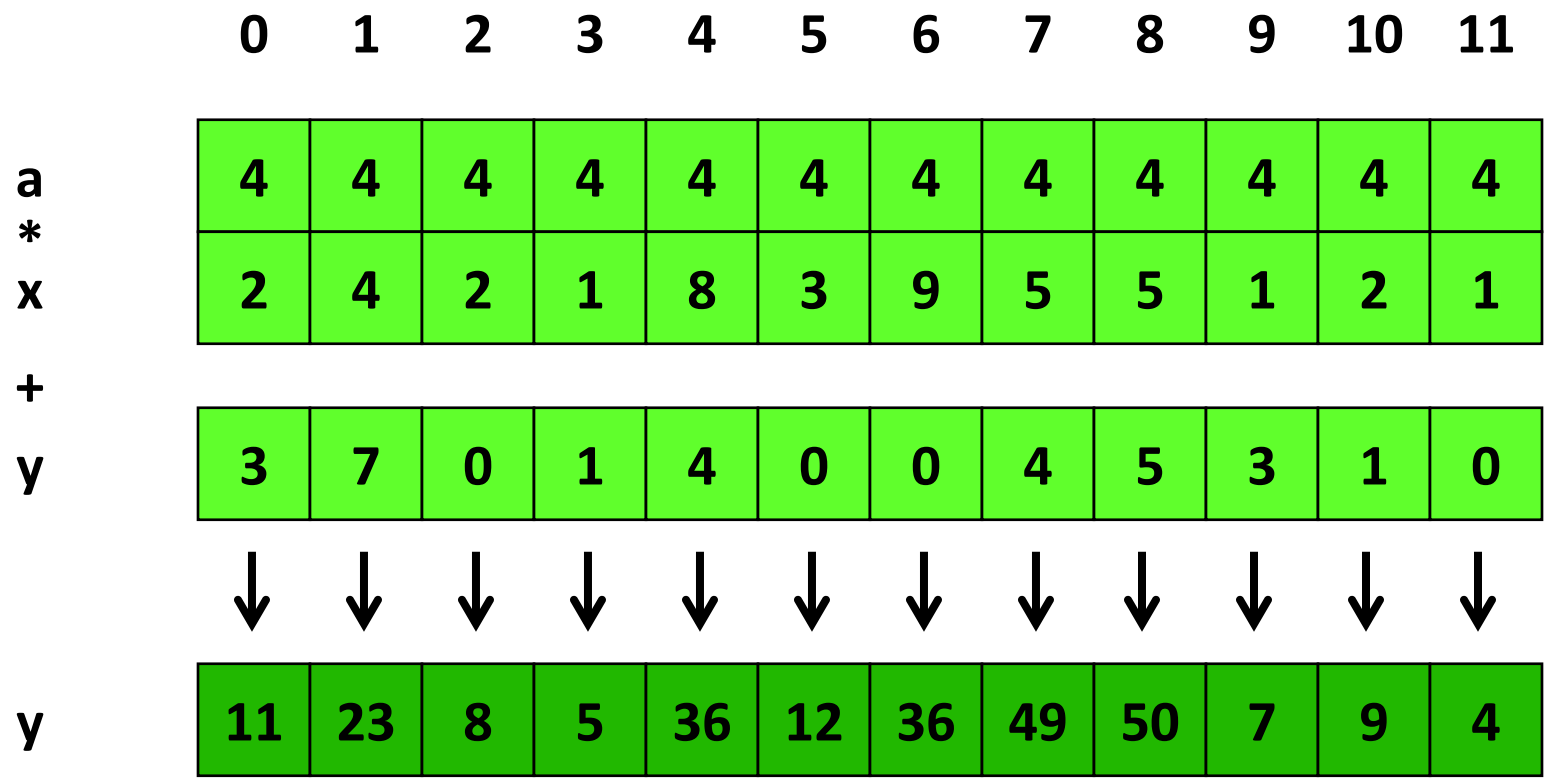


- Applies if a problem can be divided into smaller sub-problems recursively until a base case is reached that can be solved serially

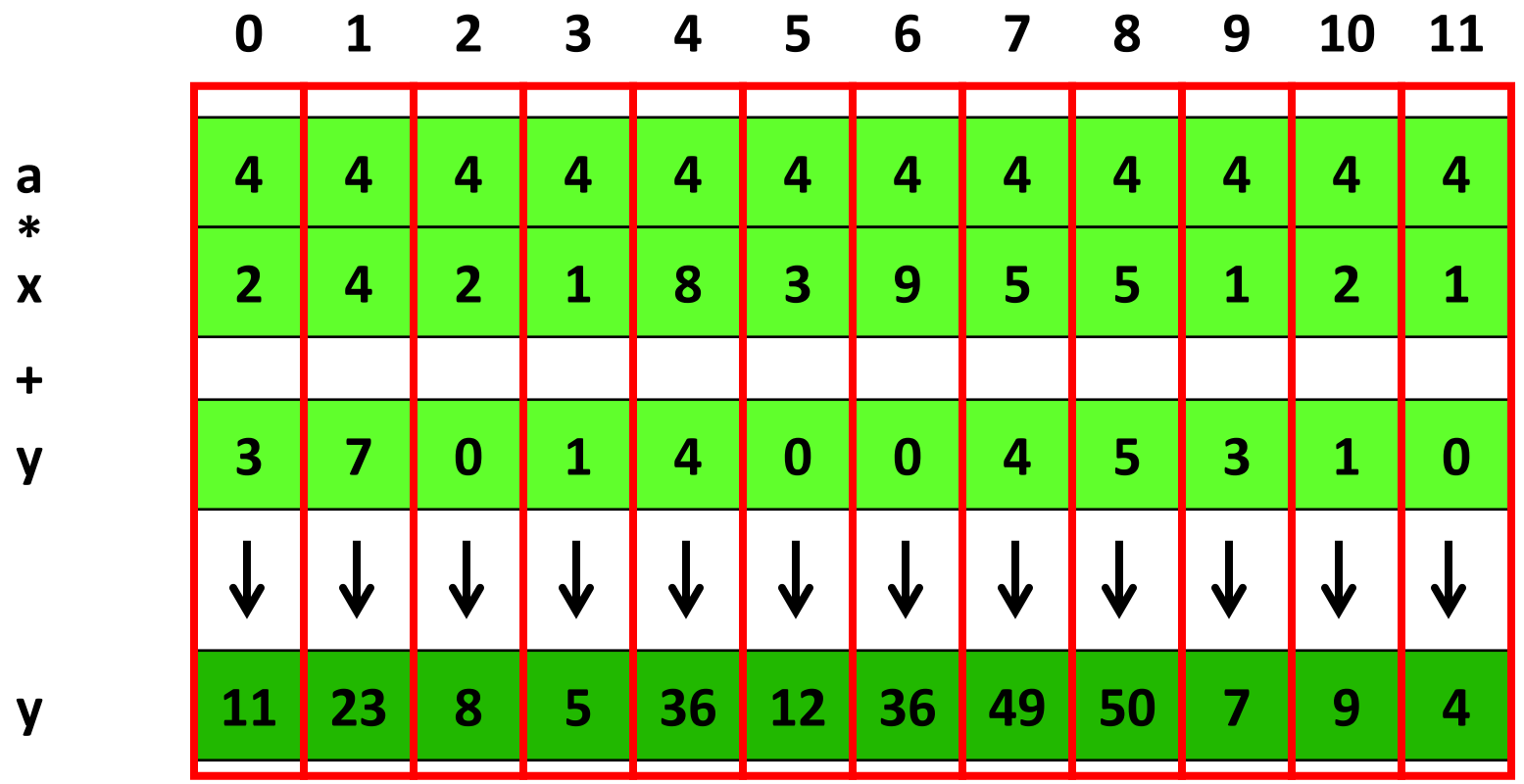
Example: Scaled Vector Addition (SAXPY)

- $y \leftarrow ax + y$
  - Scales vector x by a and adds it to vector y
  - Result is stored in input vector y
- Comes from 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?

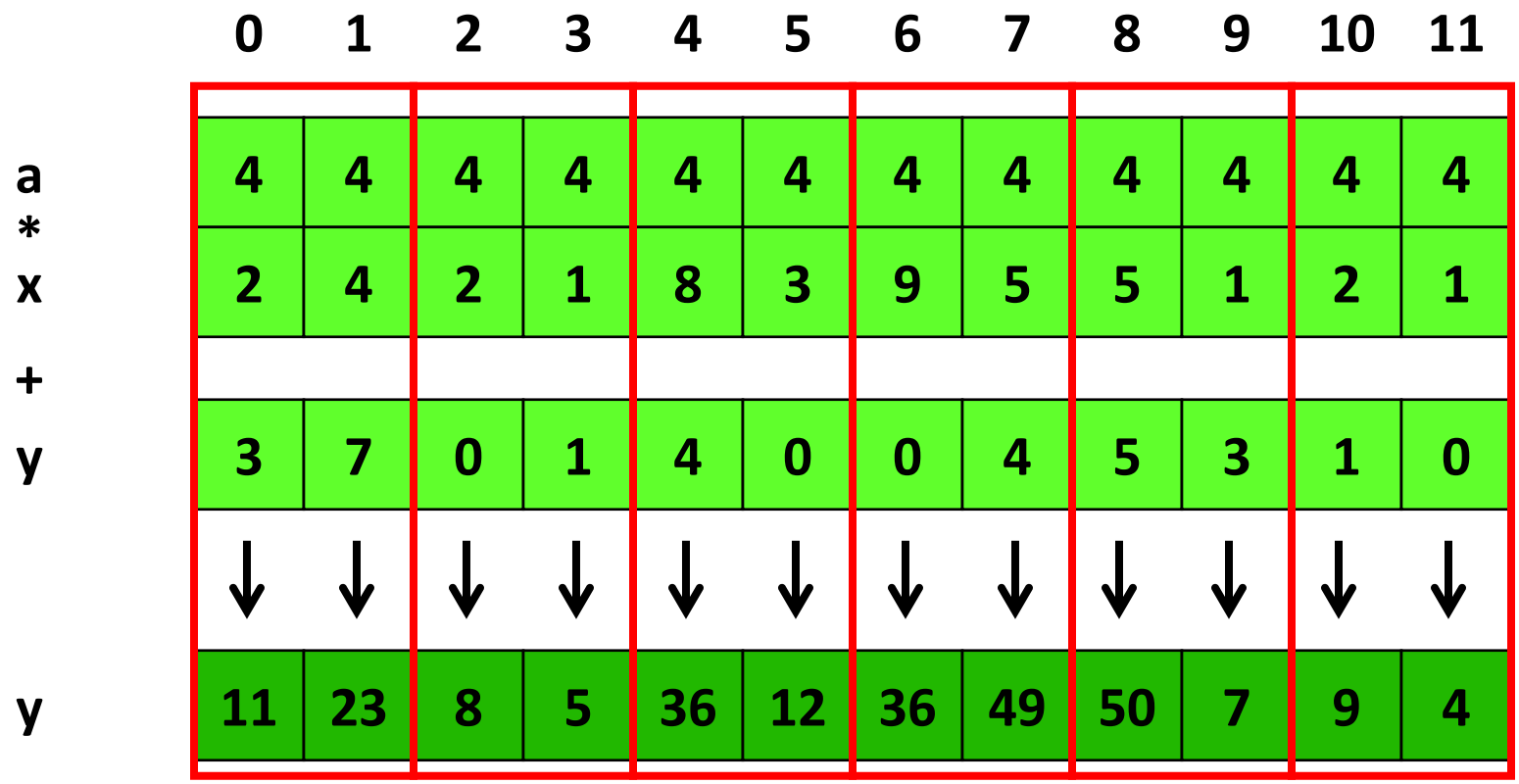


Visual: $y \leftarrow ax + y$



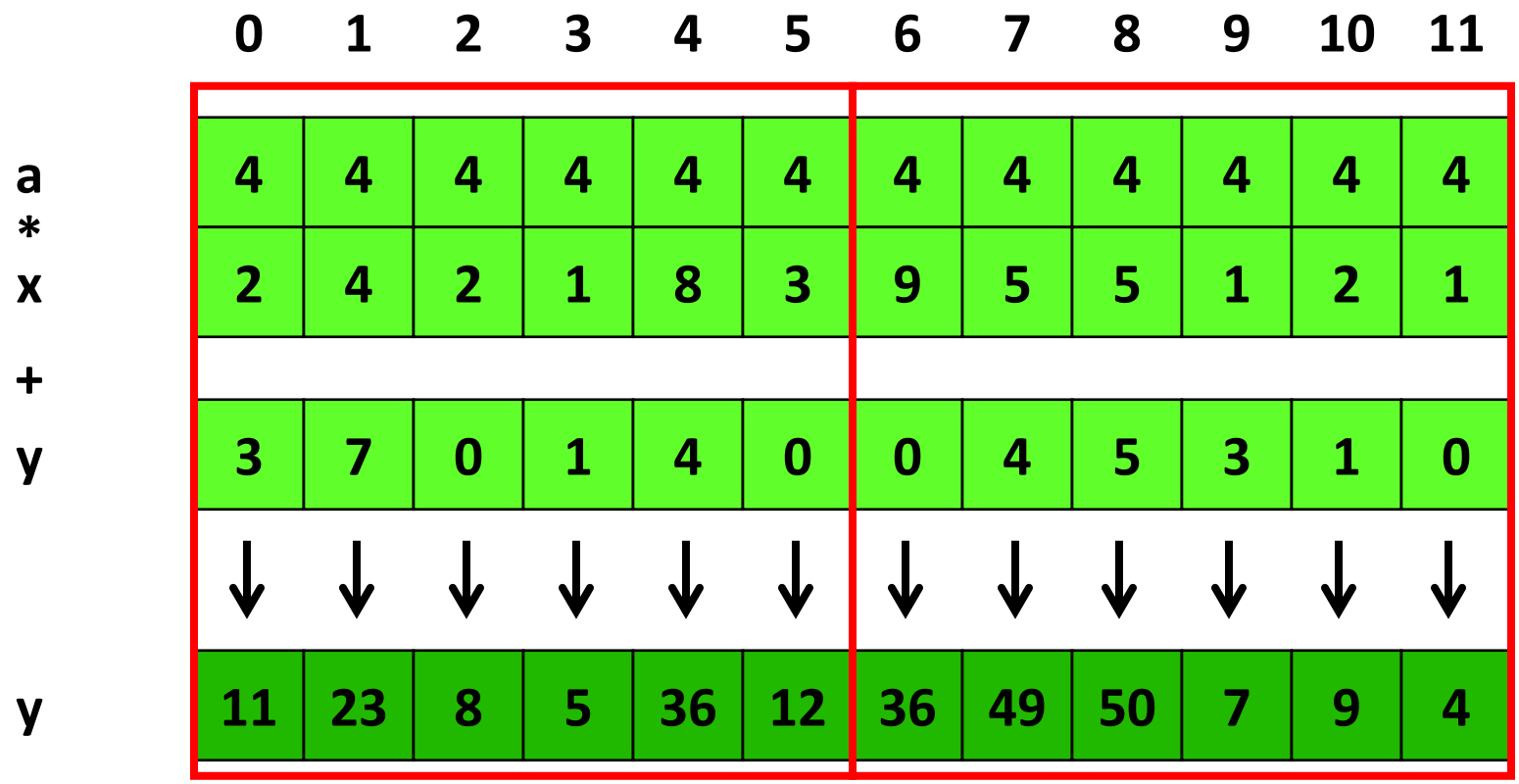
Twelve processors used → one for each element in the vector

Visual: $y \leftarrow ax + y$



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

Visual: $y \leftarrow ax + y$



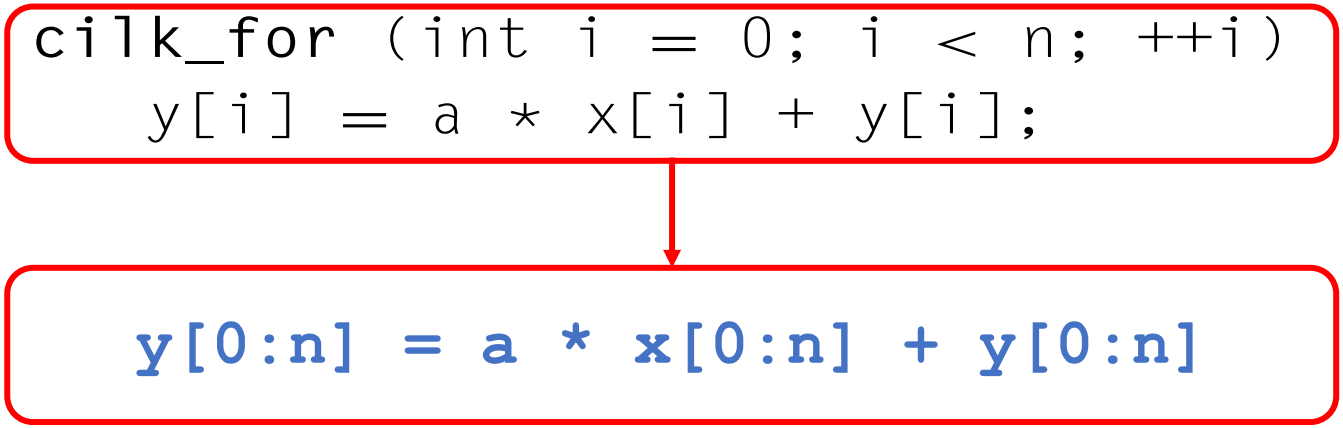
Two processors used → one for every six elements in the vector

Serial SAXPY Implementation

```
1  void saxpy_serial(
2      size_t n,           // the number of elements in the vectors
3      float a,            // scale factor
4      const float x[],    // the first input vector
5      float y[]           // the output vector and second input vector
6  ) {
7      for (size_t i = 0; i < n; ++i)
8          y[i] = a * x[i] + y[i];
9  }
```

Cilk Plus SAXPY Implementation

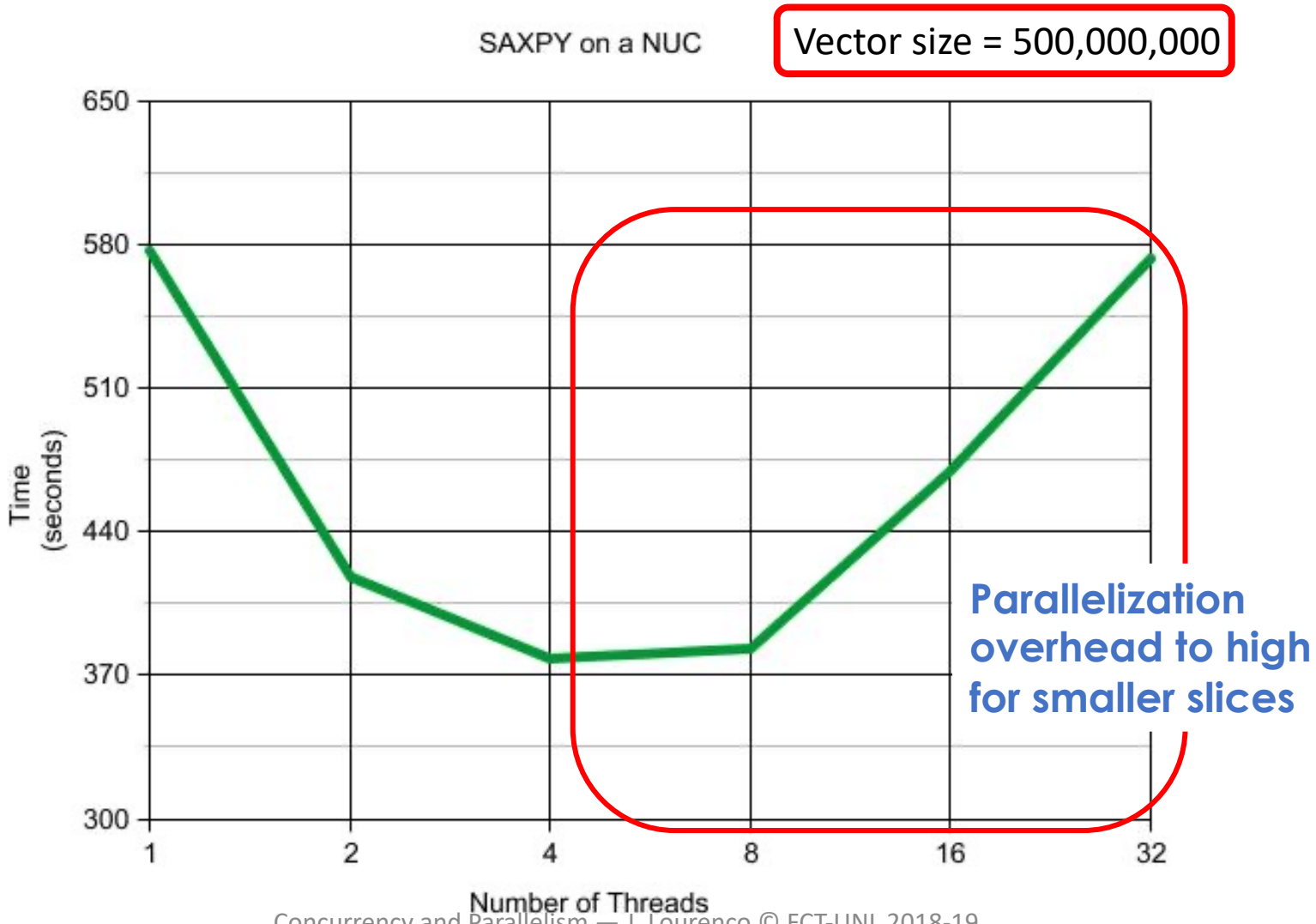
```
1  void saxpy_cilk(
2      int n,        // the number of elements in the vectors
3      float a,      // scale factor
4      float x[],    // the first input vector
5      float y[]     // the output vector and second input vector
6  ) {
7      cilk_for (int i = 0; i < n; ++i)
8          y[i] = a * x[i] + y[i];
9  }
```



OpenMP SAXPY Implentation

```
1  void saxpy_openmp(
2      int n,       // the number of elements in the vectors
3      float a,     // scale factor
4      float x[],   // the first input vector
5      float y[]    // the output vector and second input vector
6  ) {
7  #pragma omp parallel for
8      for (int i = 0; i < n; ++i)
9          y[i] = a * x[i] + y[i];
10 }
```

OpenMP SAXPY Performance

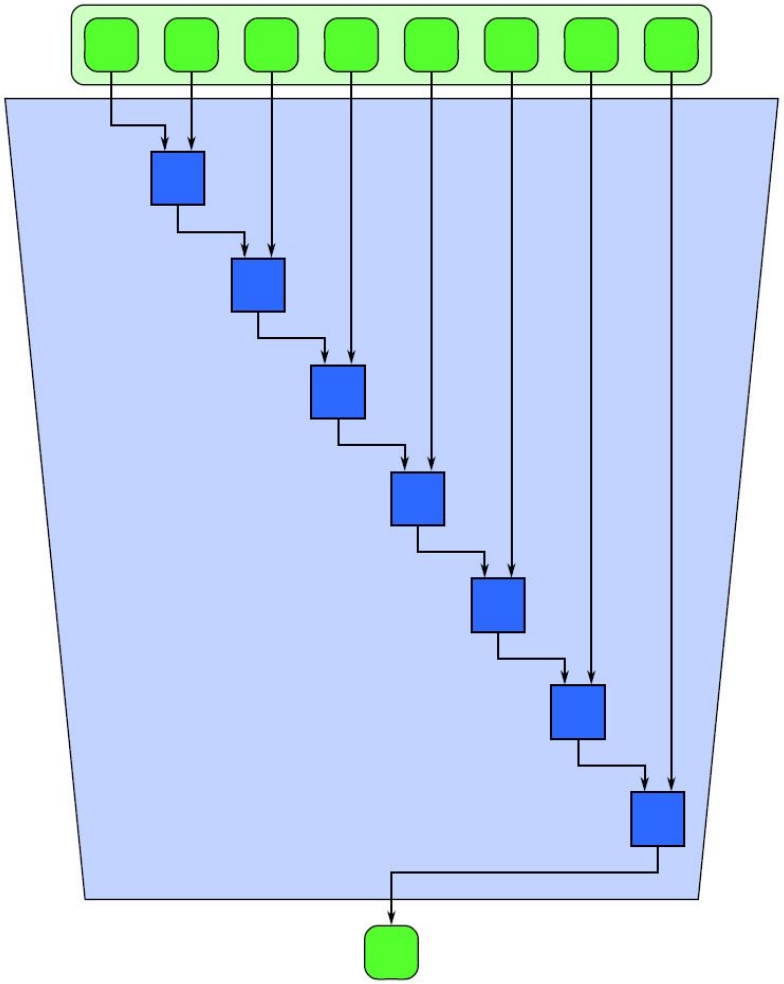


Reduce

- **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

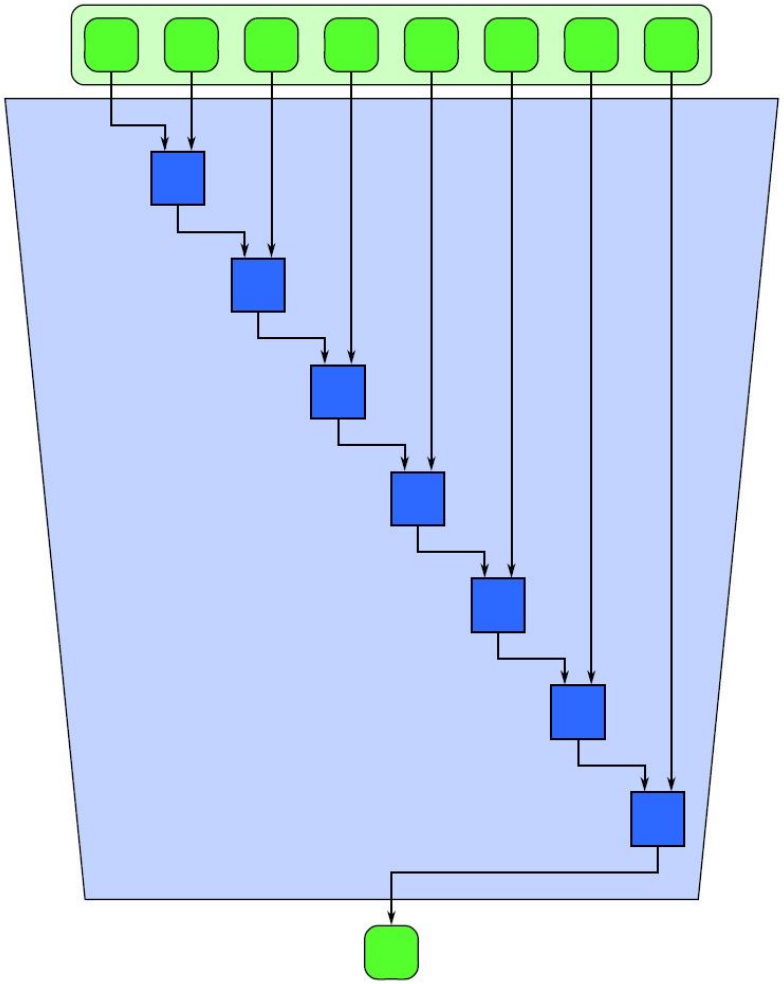


```
1  template<typename T>
2  T reduce(
3      T (*f)(T,T), // combiner function
4      size_t n, // number of elements in input array
5      T a[]     // input array
6  ) {
7      assert(n > 0);
8      T accum = a[0];
9      for (size_t i = 1; i < n; i++) {
10         accum = f(accum, a[i]);
11     }
12     return accum;
13 }
```

The input array cannot be empty!

Reduce

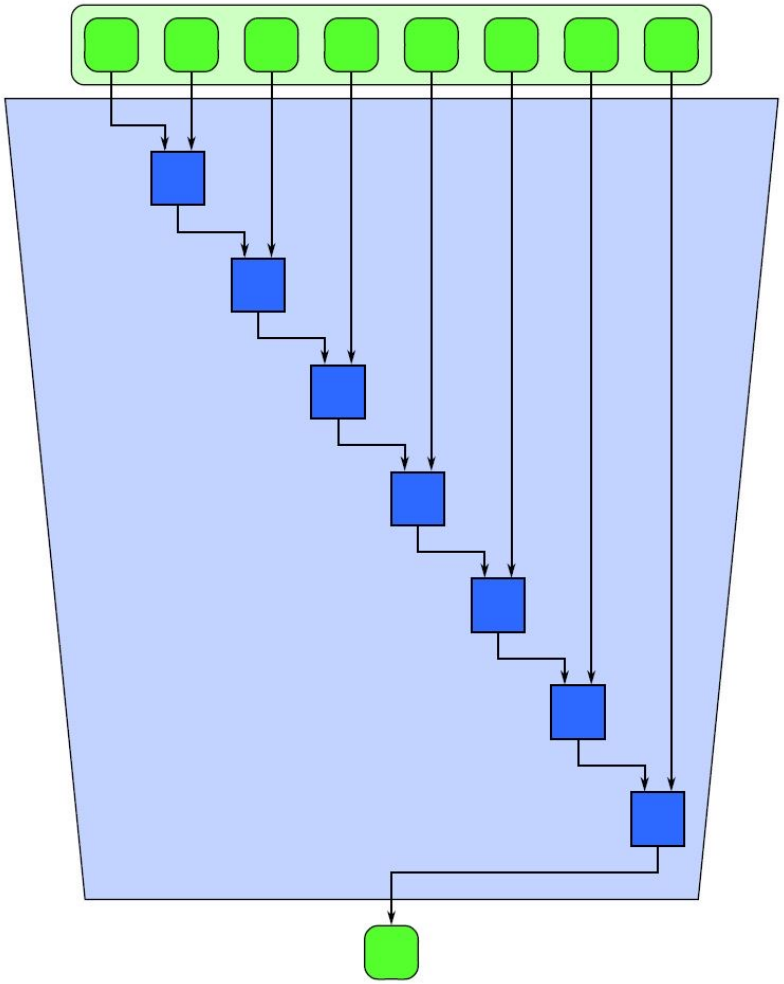
Serial Reduction



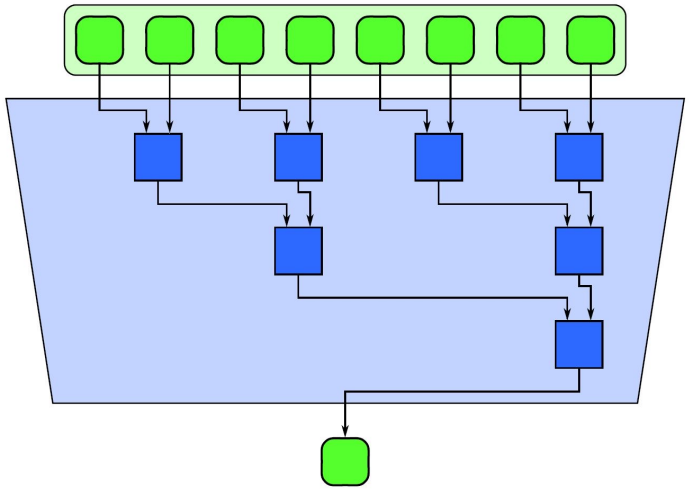
```
1  template<typename T>
2  T reduce(
3      T (*f)(T,T), // combiner function
4      size_t n, // number of elements in input array
5      T a[],    // input array
6      T identity // identity of combiner function
7  ) {
8      T accum = identity;
9      for (size_t i = 0; i < n; ++i) {
10         accum = f(accum, a[i]);
11     }
12     return accum;
13 }
```

Reduce

Serial Reduction

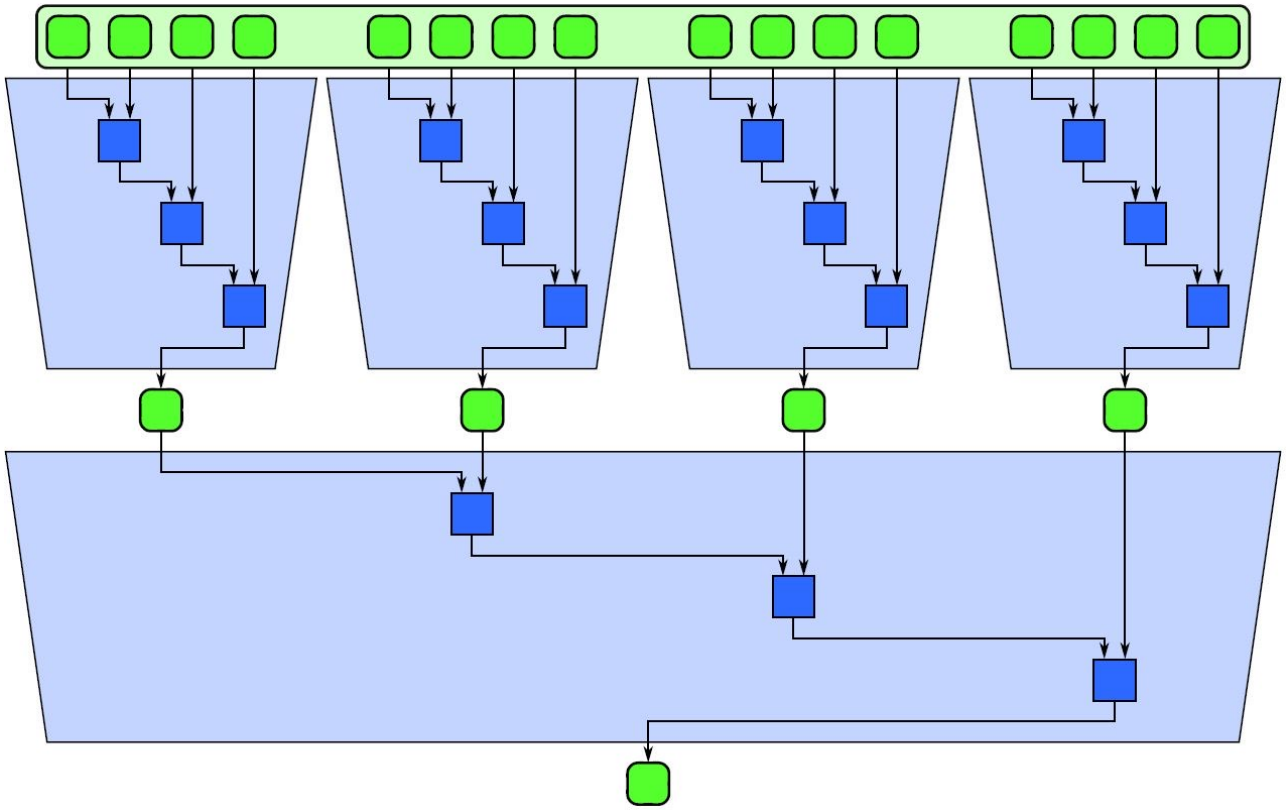


Parallel Reduction

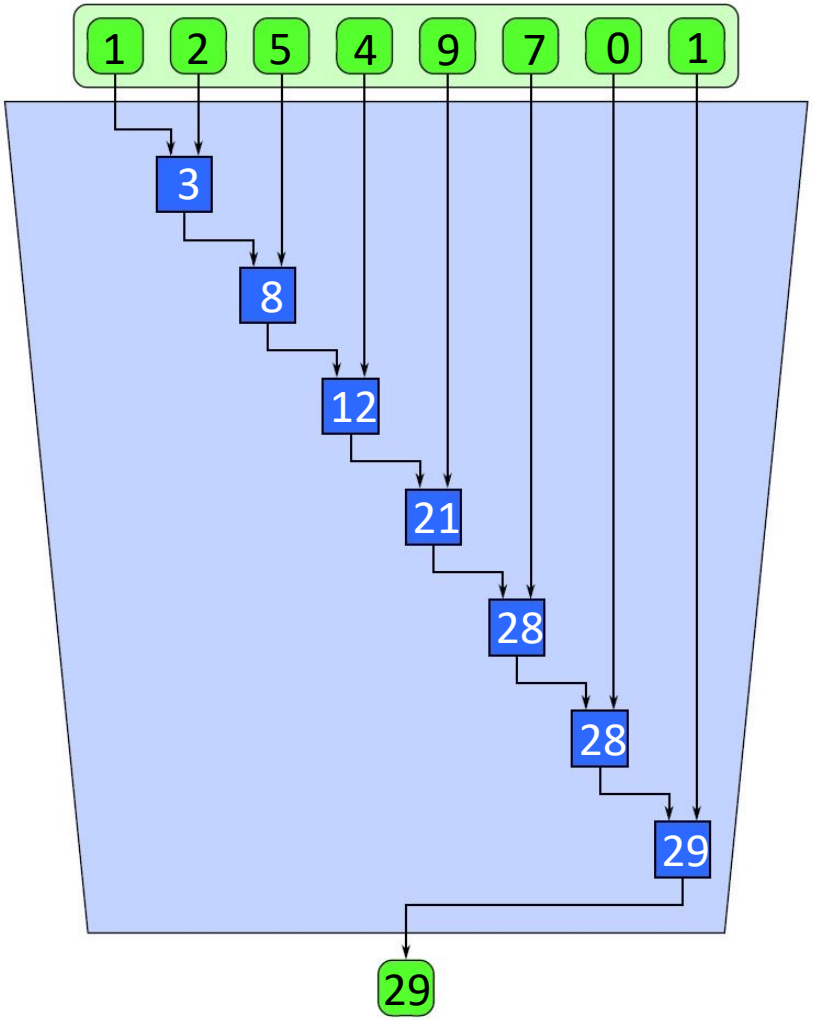


Reduce

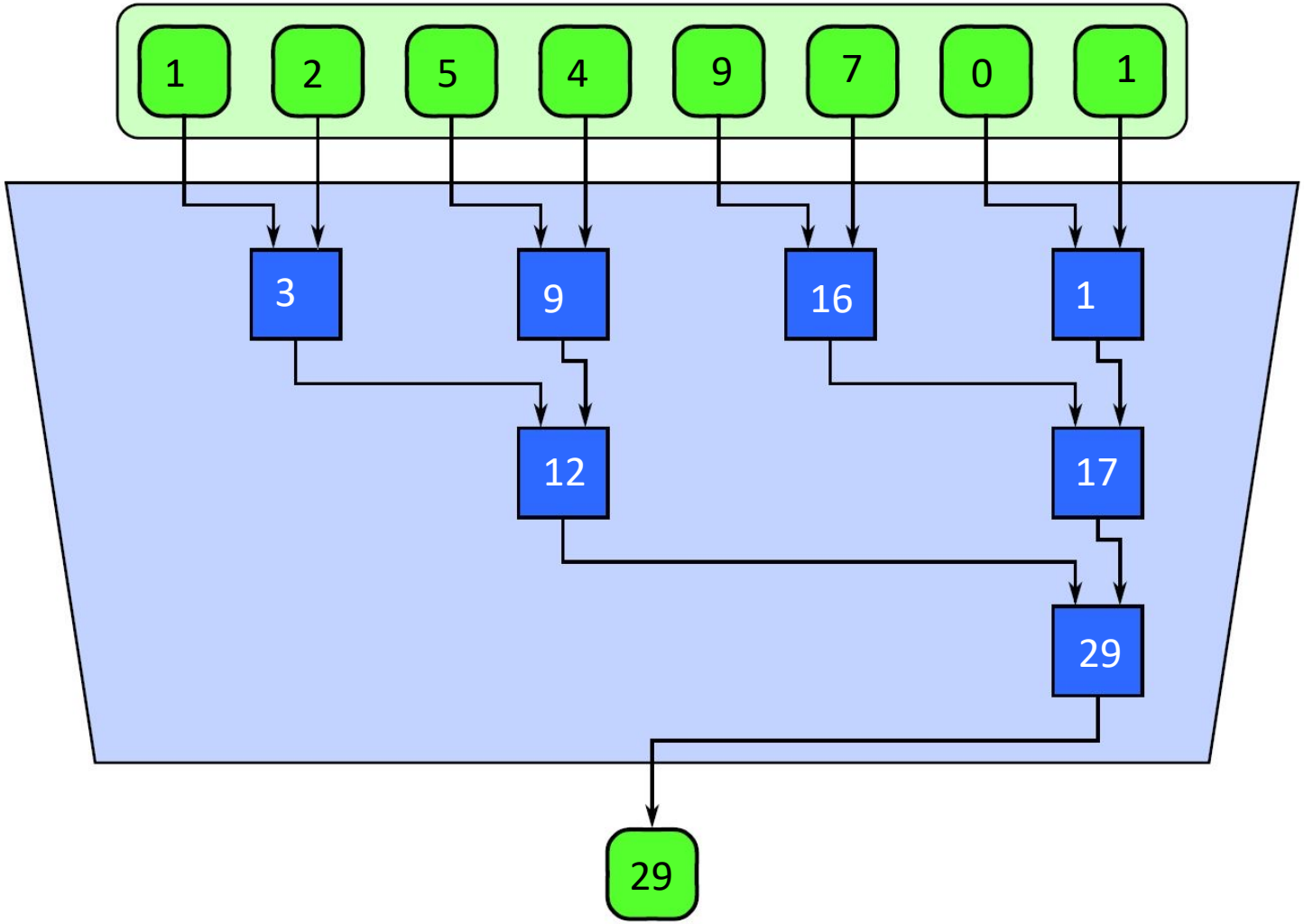
- **Tiling** is used to break chunks of work up for workers to reduce serially



Reduce – Add Example

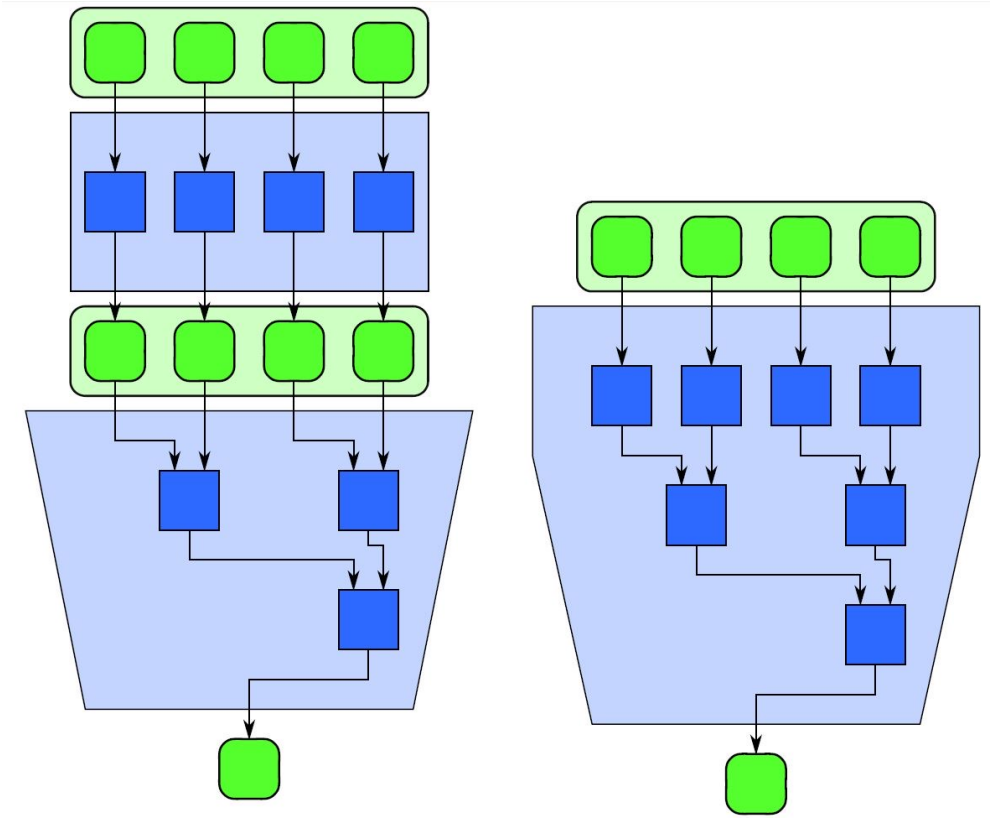


Reduce – Add Example



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

The END