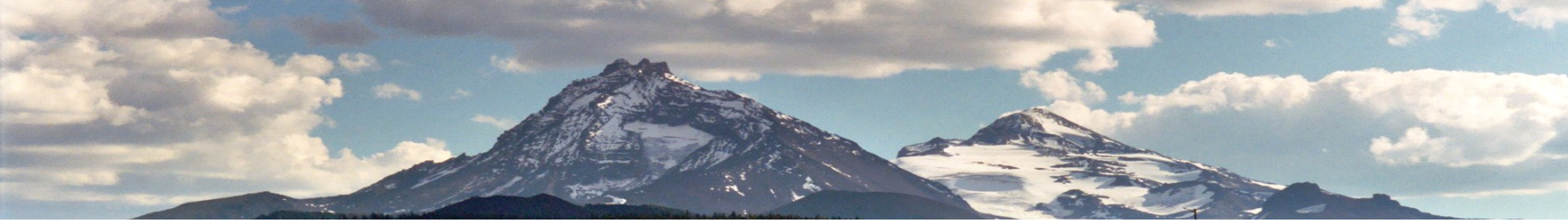


Map Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science

UNIVERSITY OF OREGON

Table of Contents

Table of Contents

Mapping

- "Do the same thing many times"

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

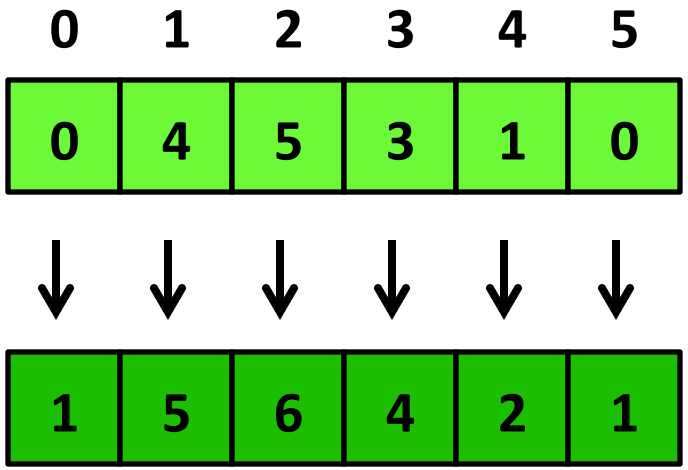
- Well-known higher order function in languages like ML, Haskell, Scala

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

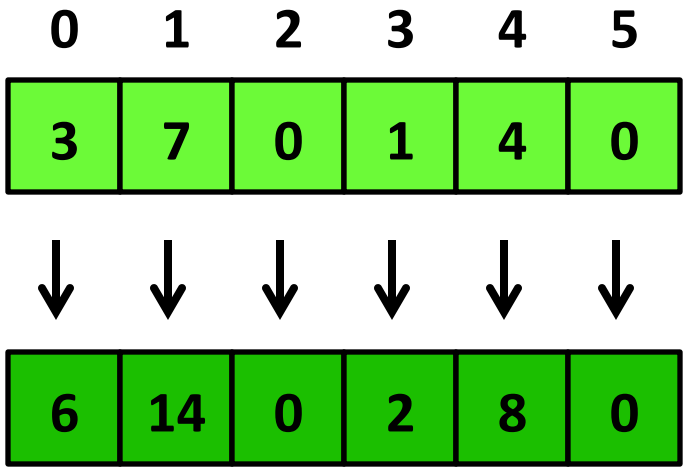
applies a function each element in a list and returns a list of results

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”

Key Idea

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

Key Idea

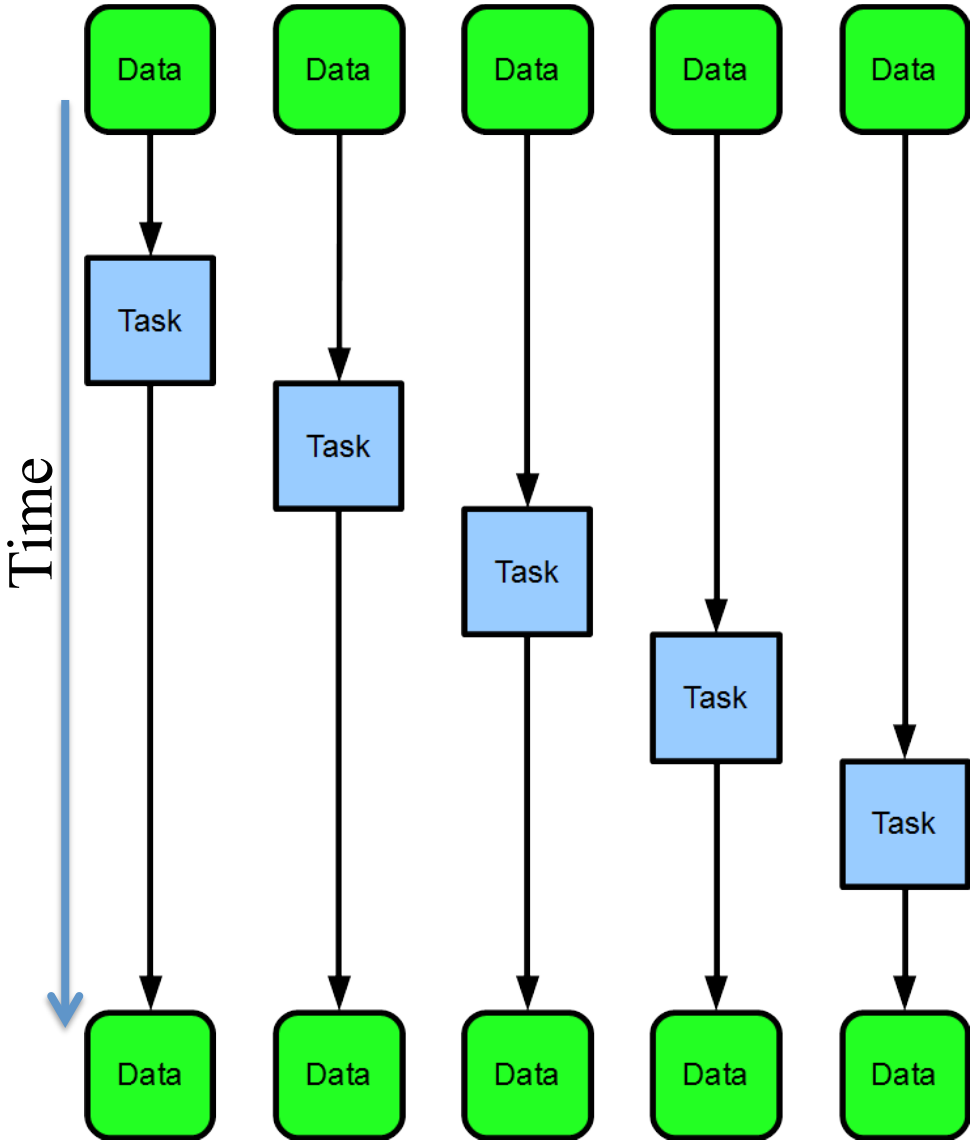
- Map is a "foreach loop" where each iteration is independent

Embarrassingly Parallel

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

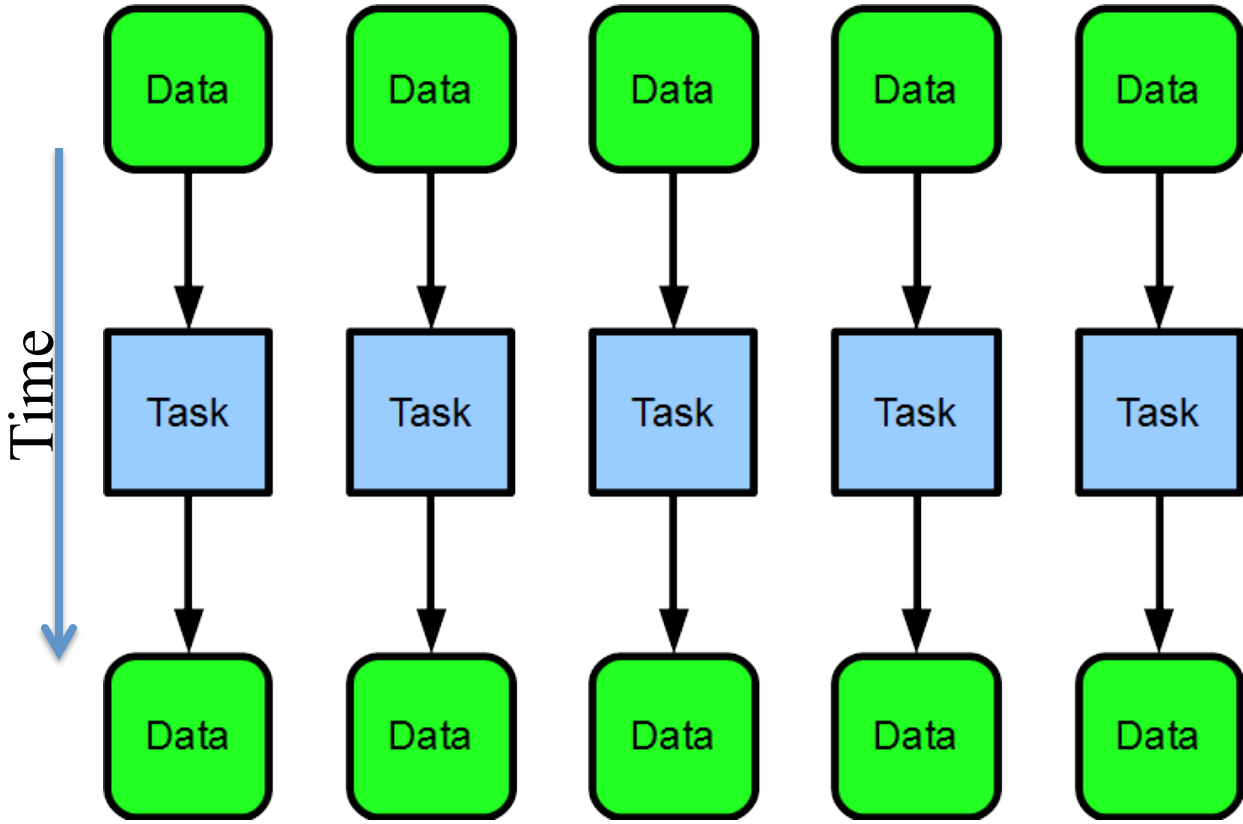
Sequential Map

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

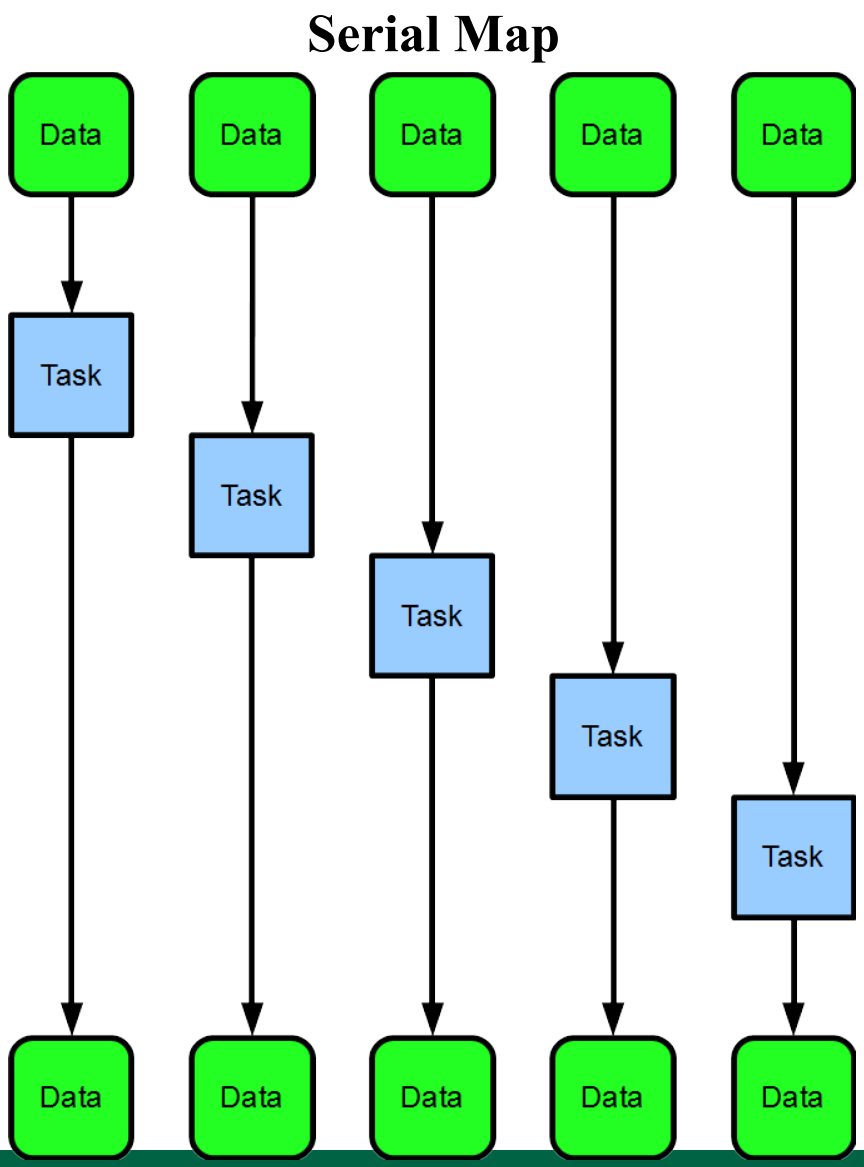


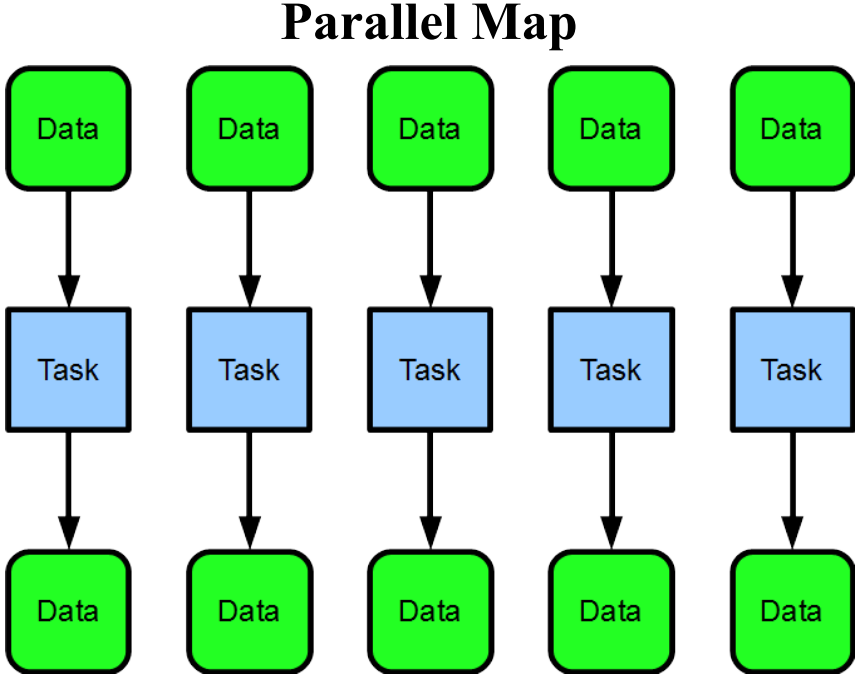
Parallel Map

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

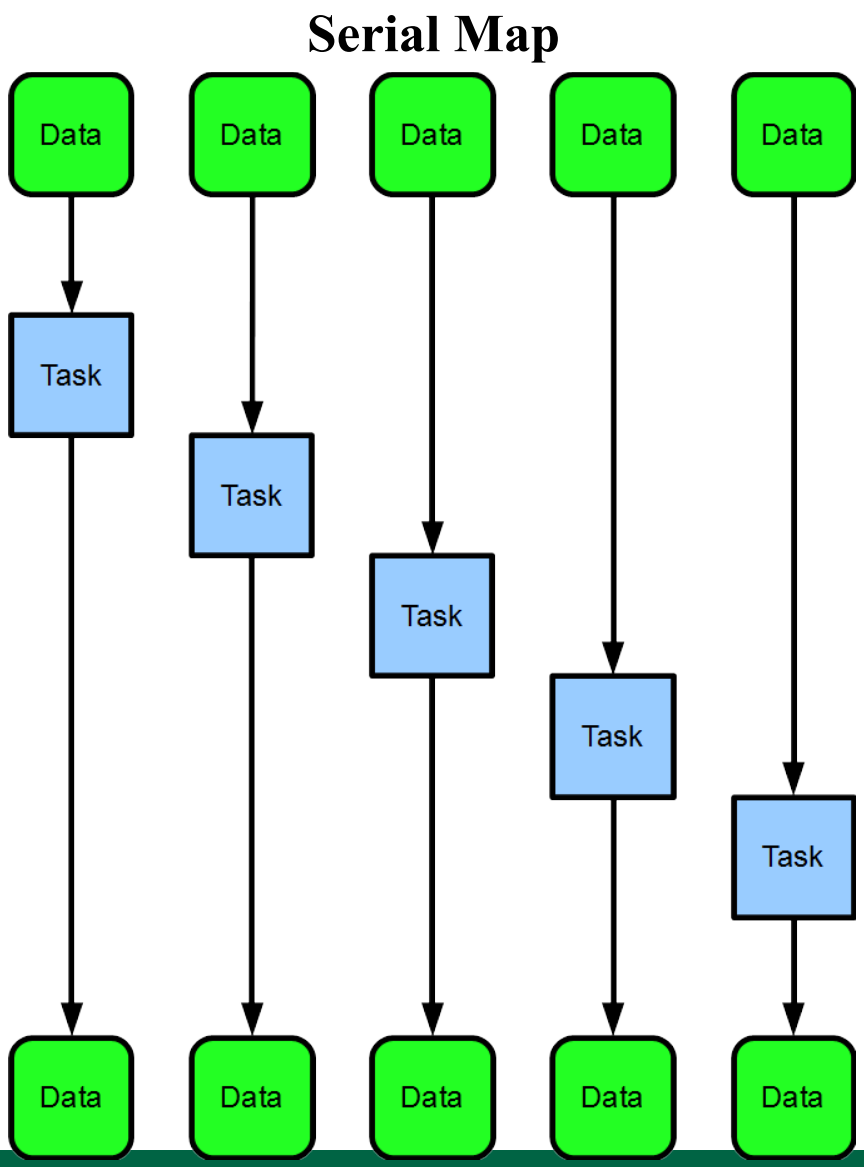


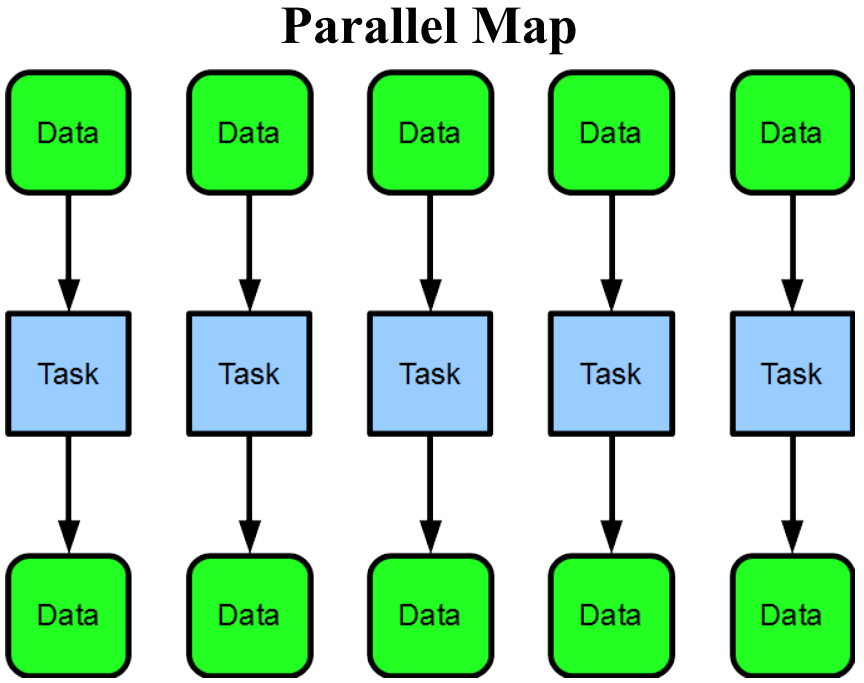
Comparing Maps





Comparing Maps





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 (embarrasing) parallelism is independence

Independence

- The key to (embarrasing) 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 CilkPlus contain a parallel ***`for`*** language construct
- Map is a mode of use of parallel ***`for`***
- TBB uses **higher order functions** with lambda expressions/"funtors"
- Some languages (CilkPlus, Matlab, Fortran) provide **array notation** which makes some maps more concise

Array Notation

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

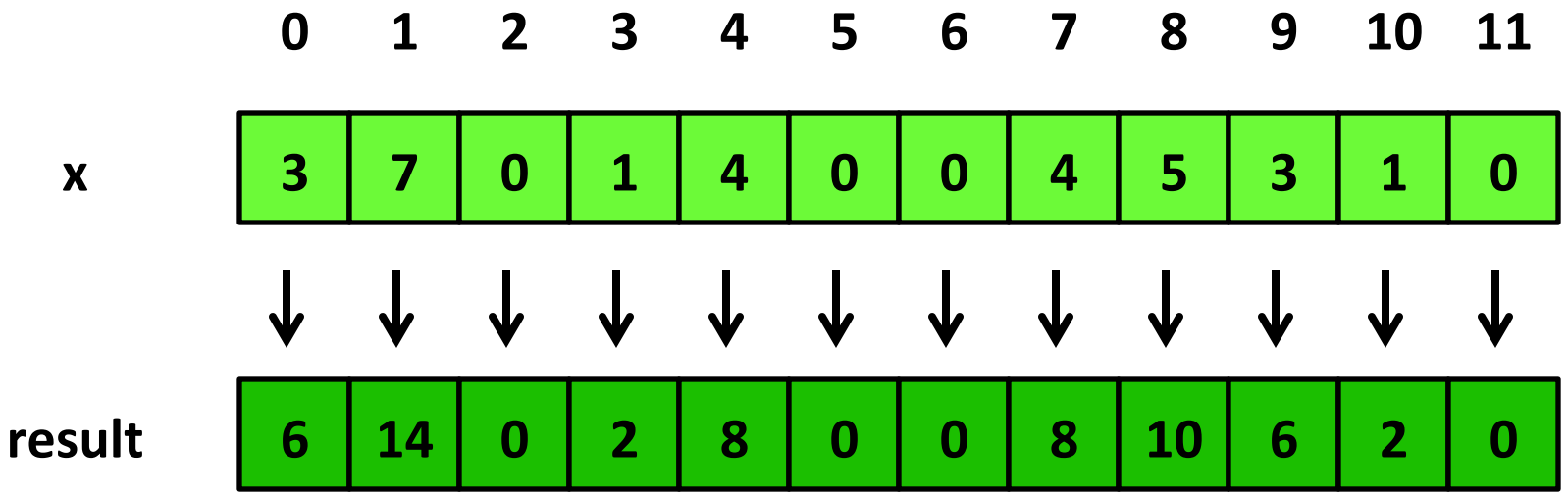
is CilkPlus array notation for "multiply every element in *A* by 5"

Unary Maps

Unary Maps

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

Map with 1 Input, 1 Output



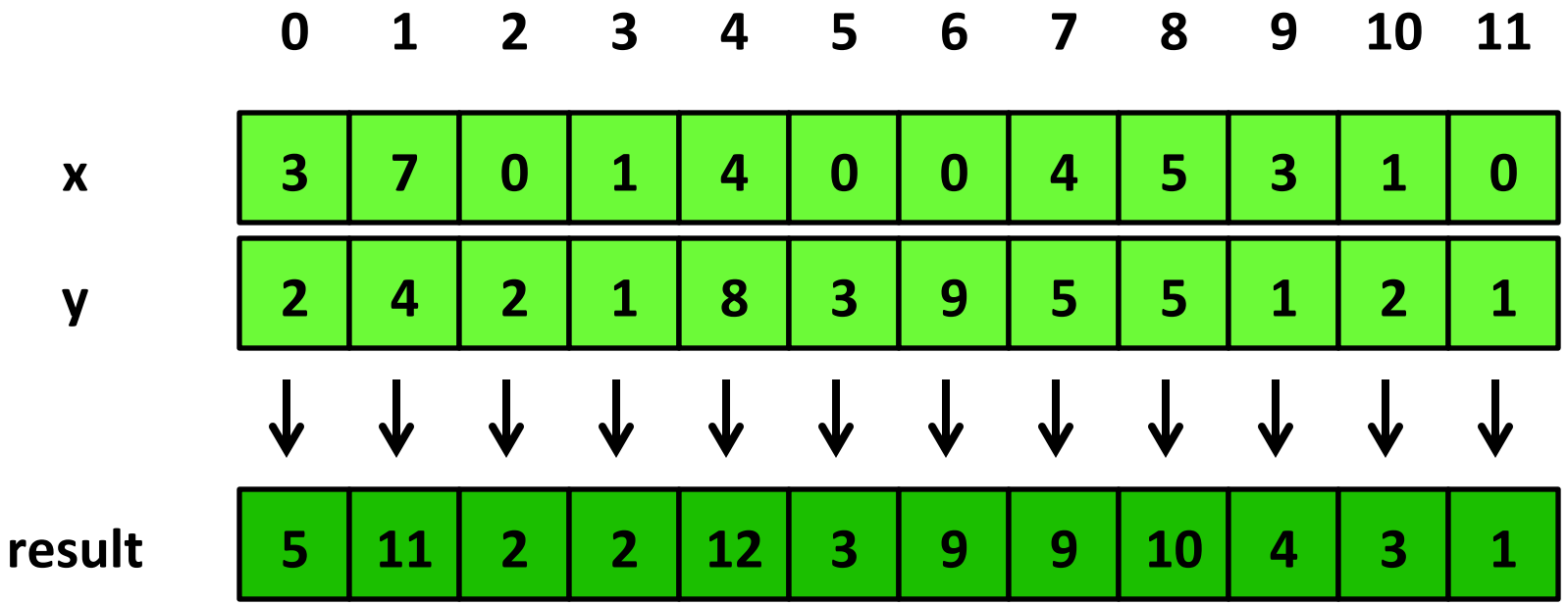

```
int  oneToOne ( int x[11] ) {
        return x*2;
}
```

N-ary Maps

N-ary Maps

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

Map with 2 Inputs, 1 Output

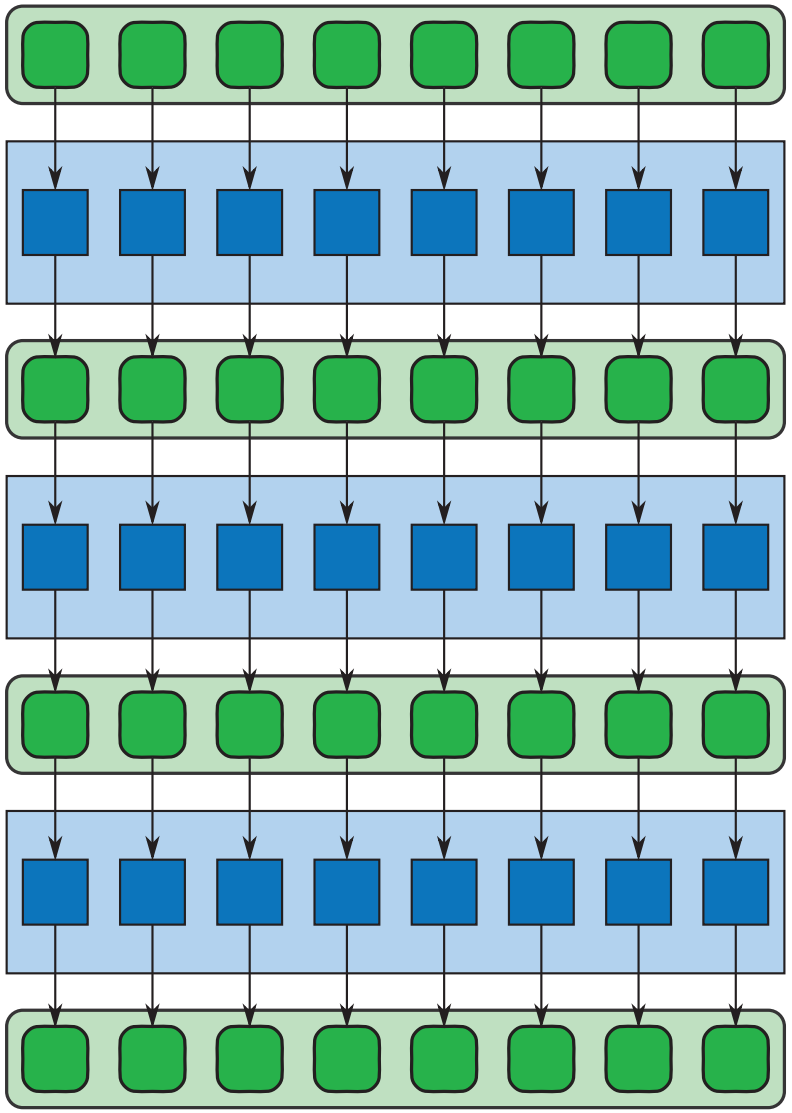



```
int  twoToOne ( int x[11], int y[11] ) {
        return x+y;
}
```

Table of Contents

Table of Contents

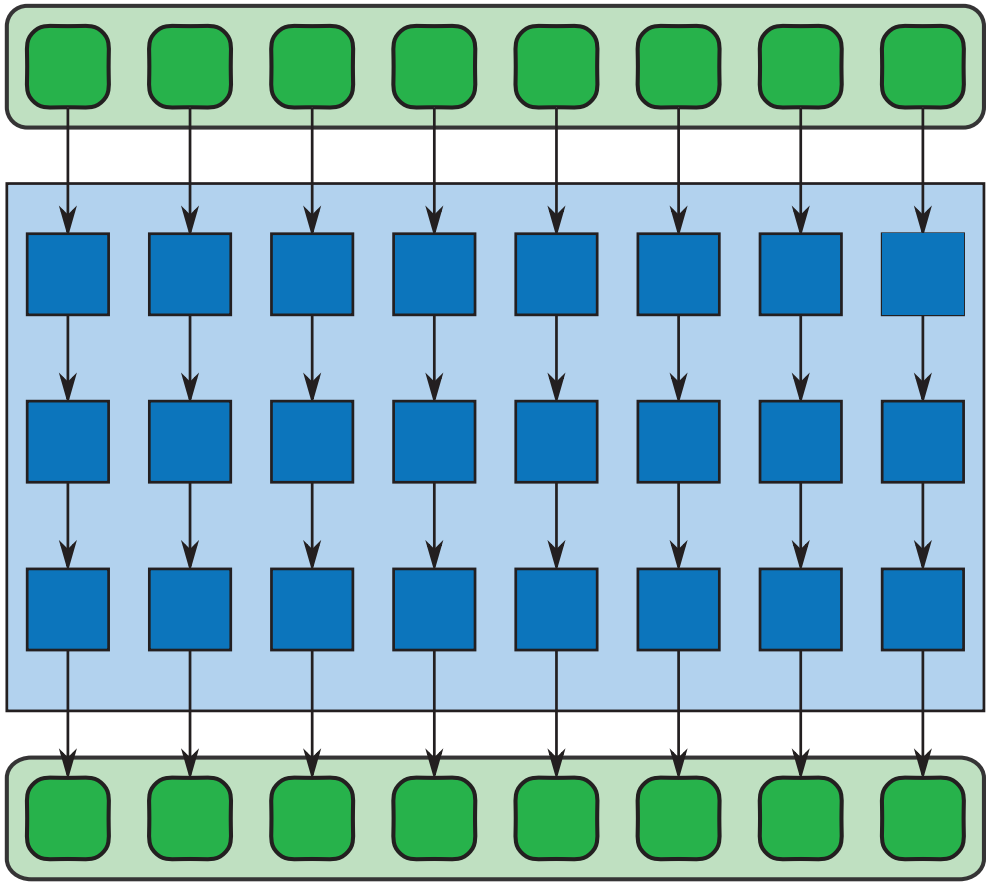
Sequences of Maps



- 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

Table of Contents

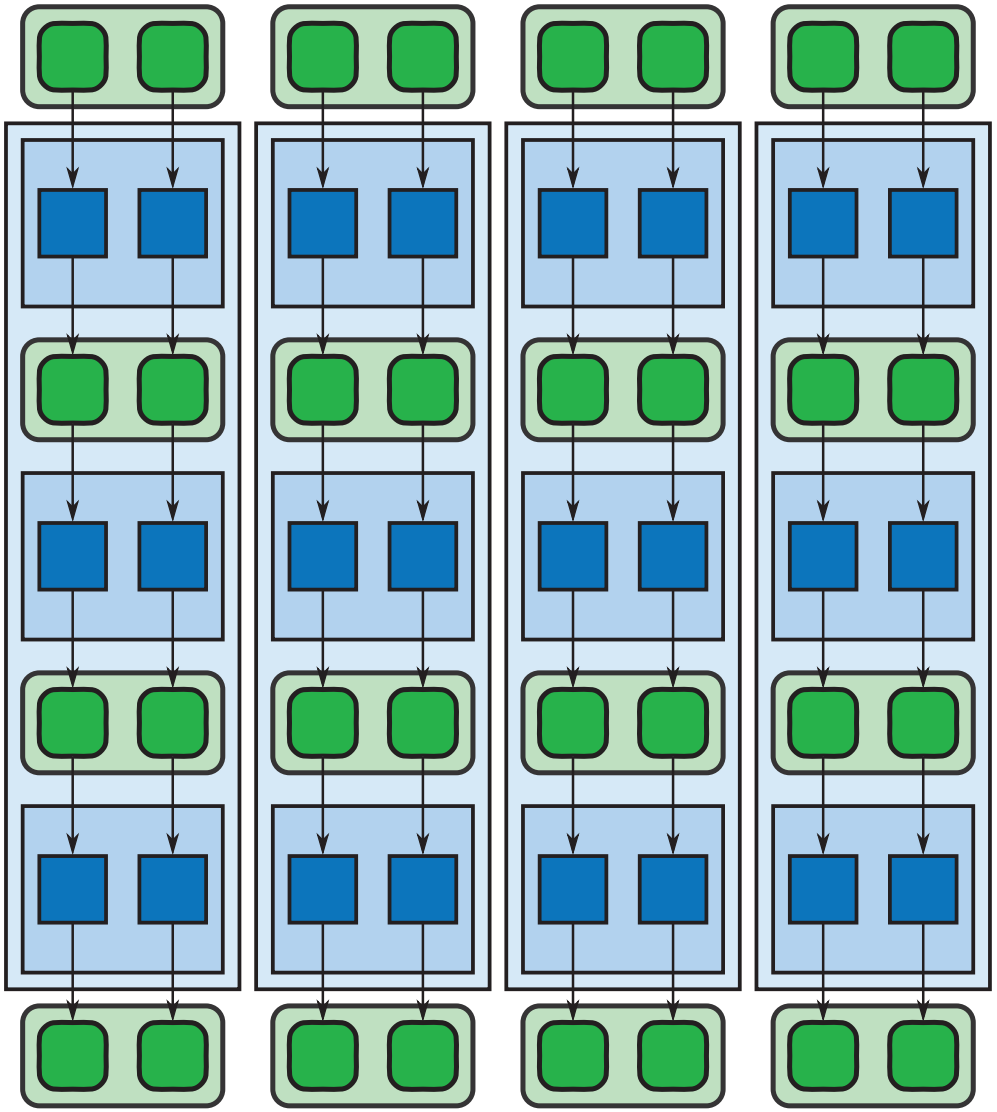
Code Fusion



- 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

Table of Contents

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

Table of Contents

Overview

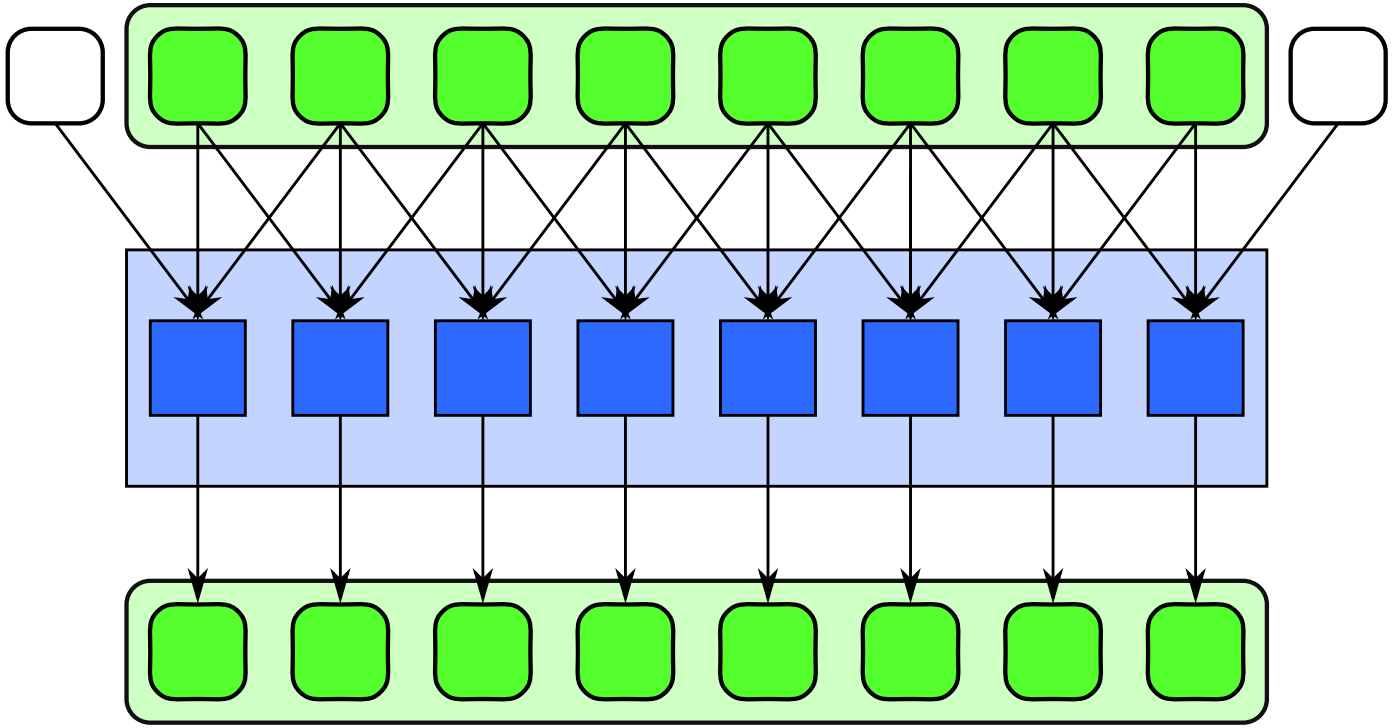
Three patterns related to map are discussed here:

- Stencil
- Workpile
- Divide-and-Conquer

They will be discussed more in detail in a later lecture.

Stencil

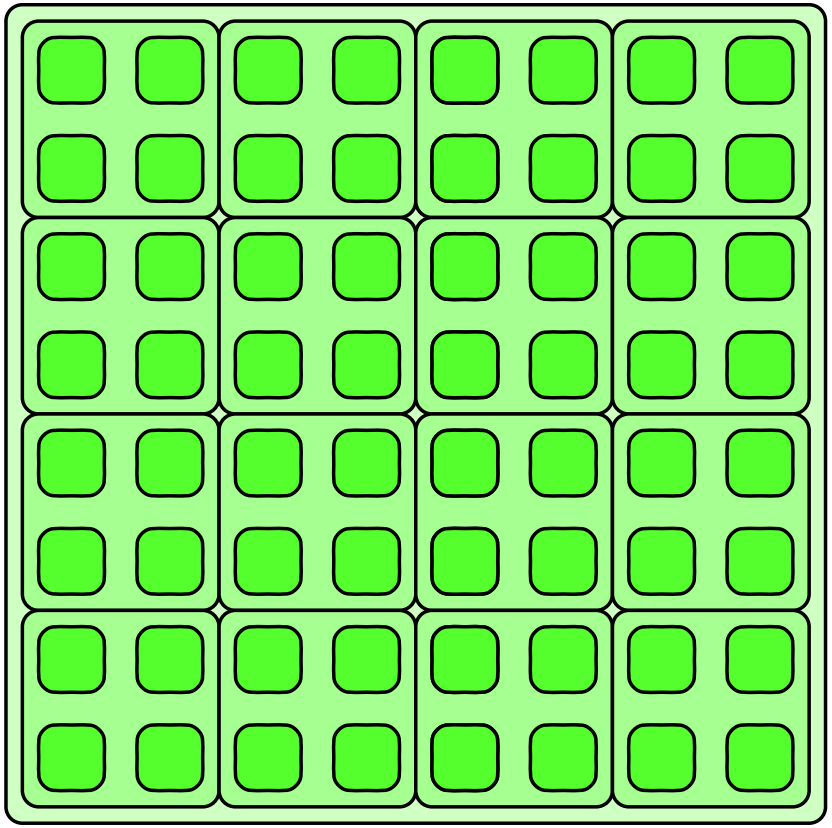
- Each instance of the map function accesses neighbors of its input, offset from its usual input
- Common in imaging and PDE solvers



Workpile

- 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 subproblems recursively until a base case is reached that can be solved serially

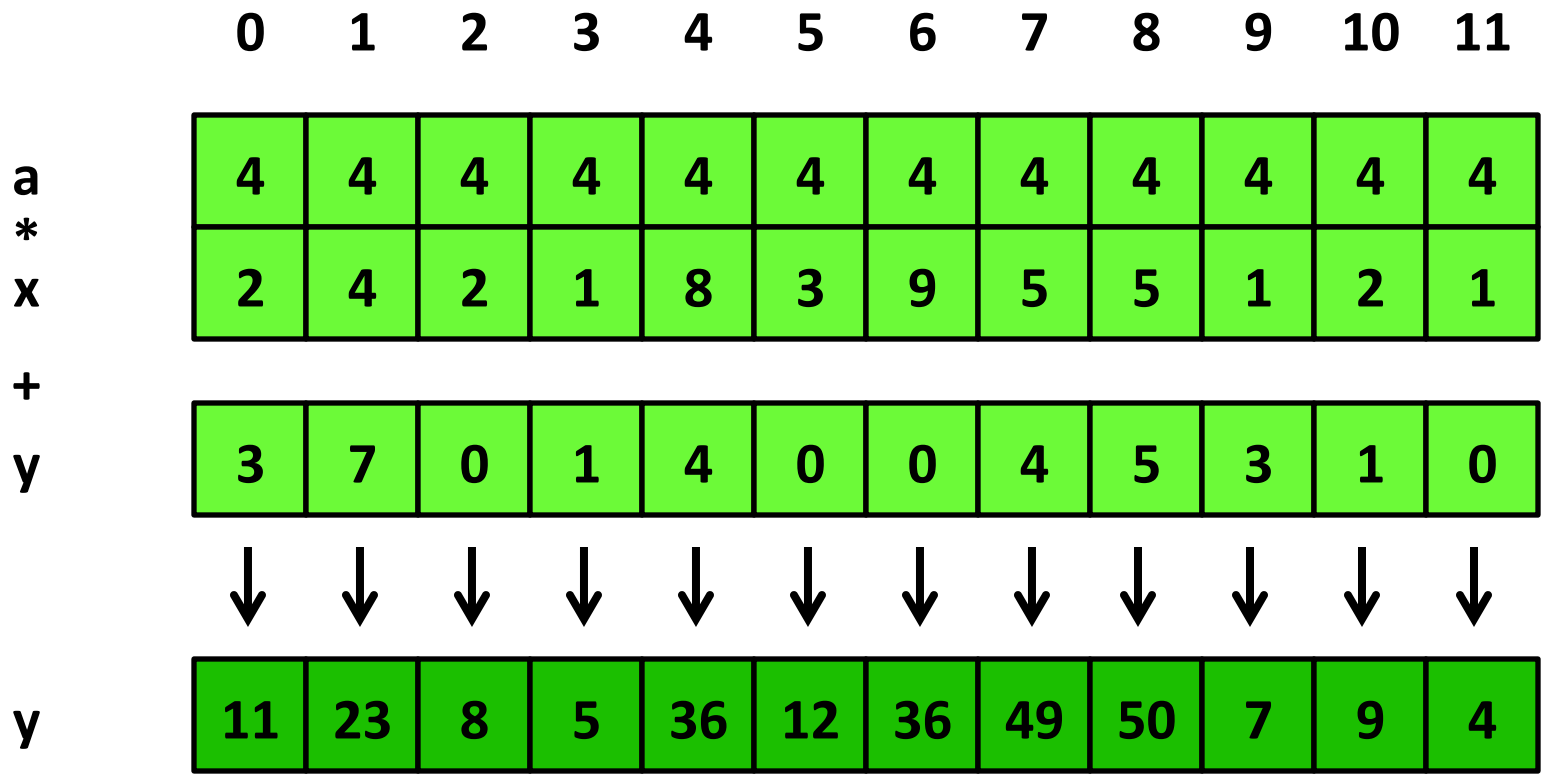
Table of Contents

Table of Contents

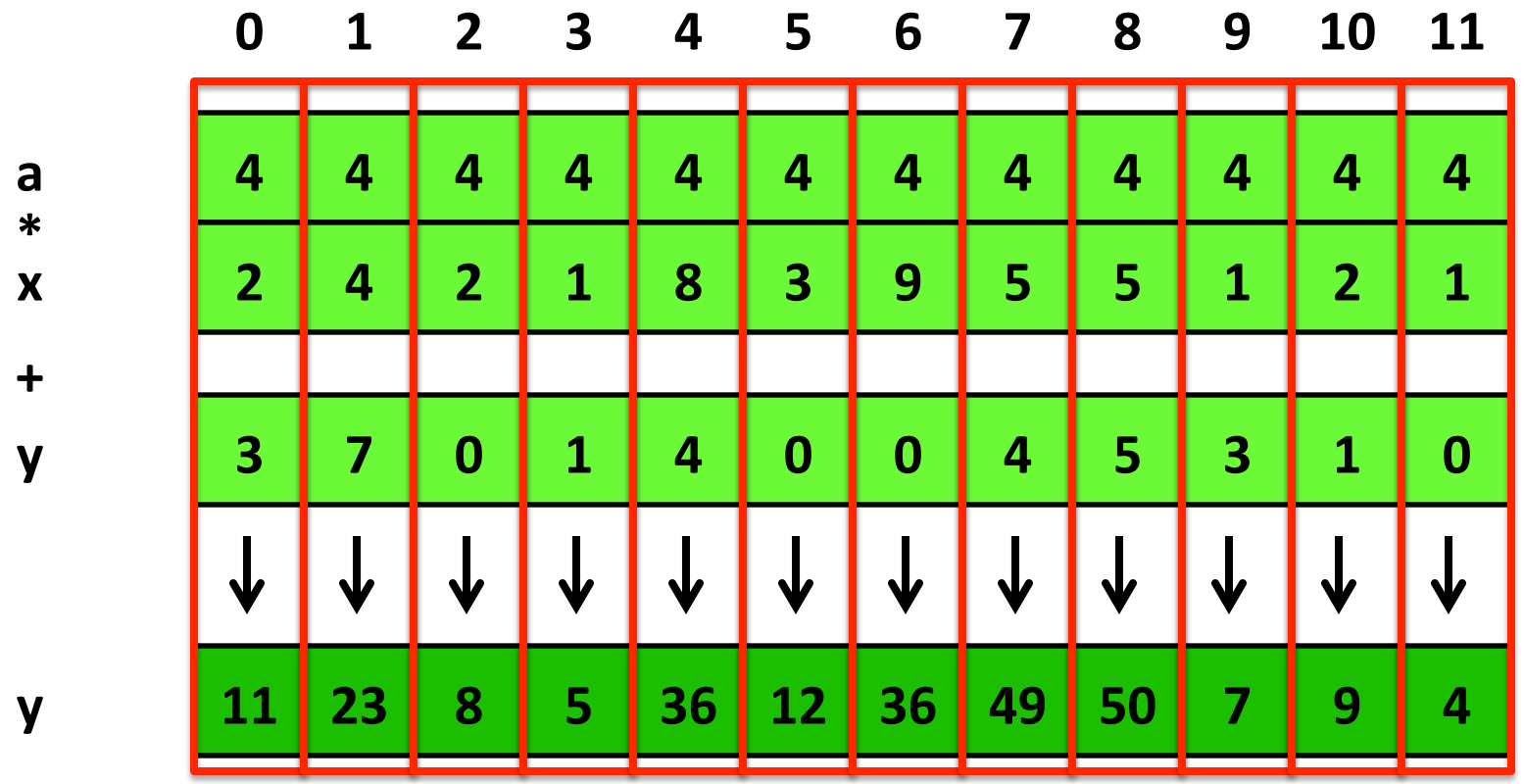
Problem Description

- $y \leftarrow ax + y$
  - 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**

Visual: $y \leftarrow ax + y$

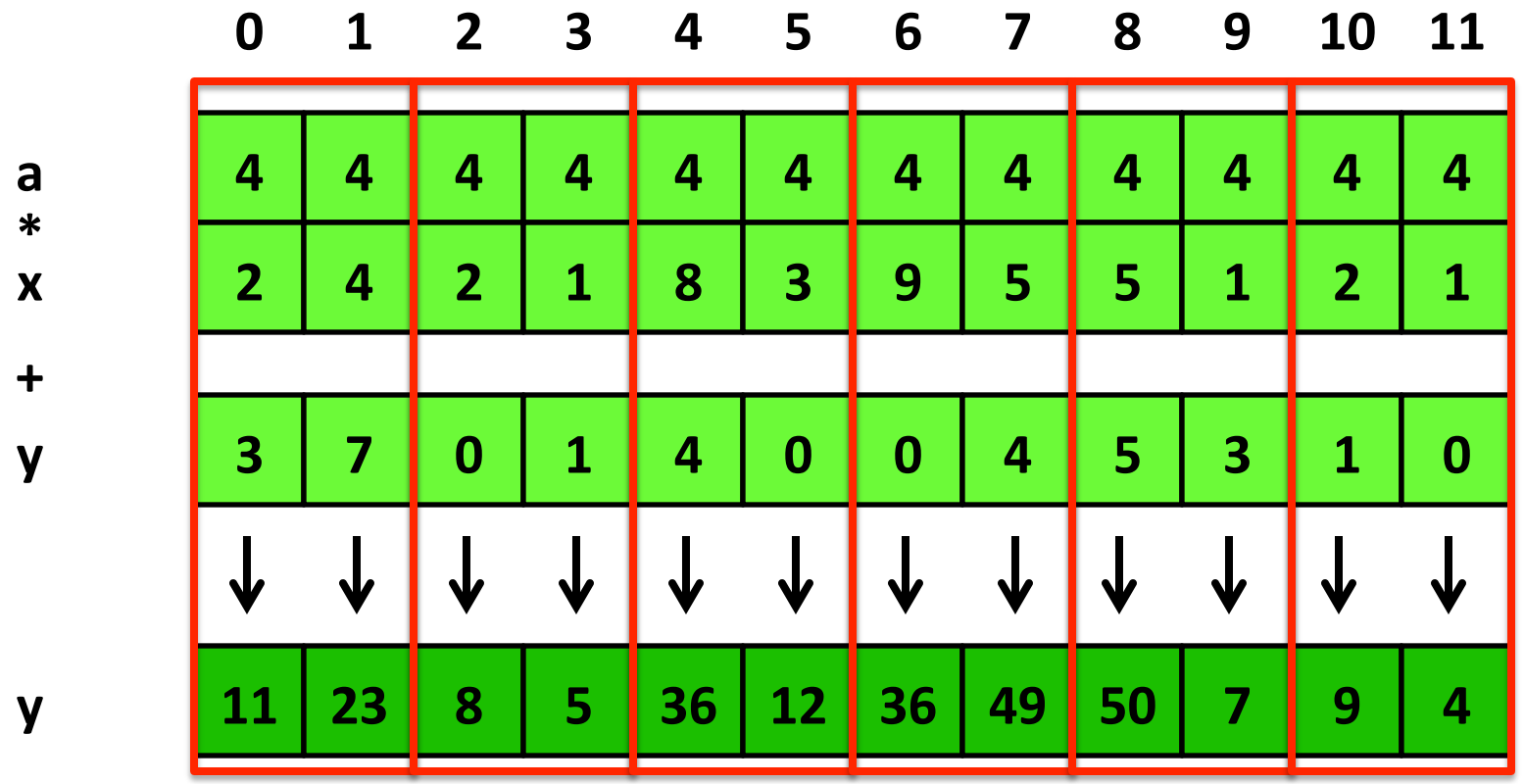


Visual: $y \leftarrow ax + y$



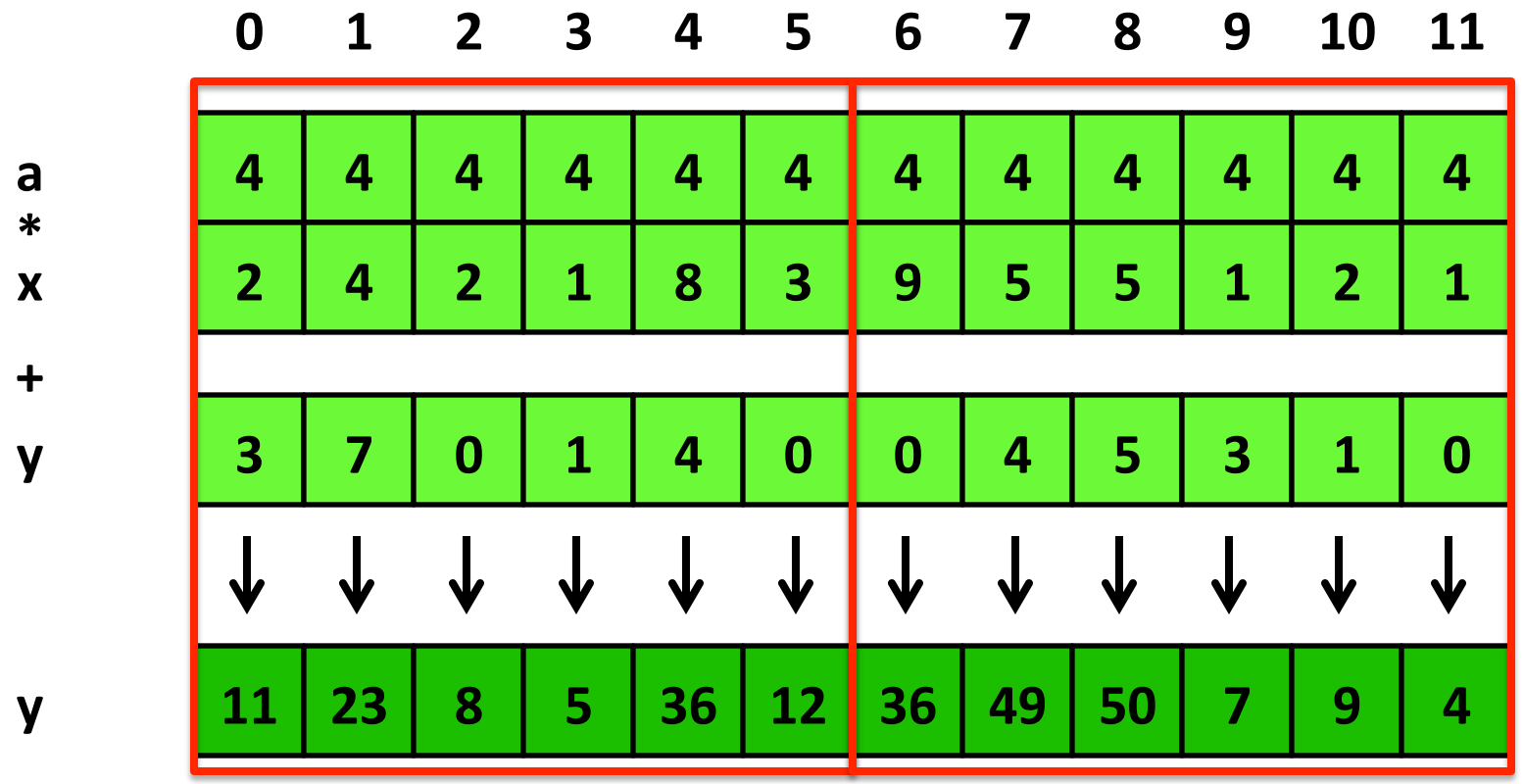
Twelve processors used → one for each element in the vector

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

Table of Contents

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  }
```

TBB SAXPY Implementation

```
1  void saxpy_tbb(
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      tbb::parallel_for(
8          tbb::blocked_range<int>(0, n),
9          [&](tbb::blocked_range<int> r) {
10             for (size_t i = r.begin(); i != r.end(); ++i)
11                 y[i] = a * x[i] + y[i];
12         }
13     );
14 }
```

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 }
```