Parallel Control Patterns

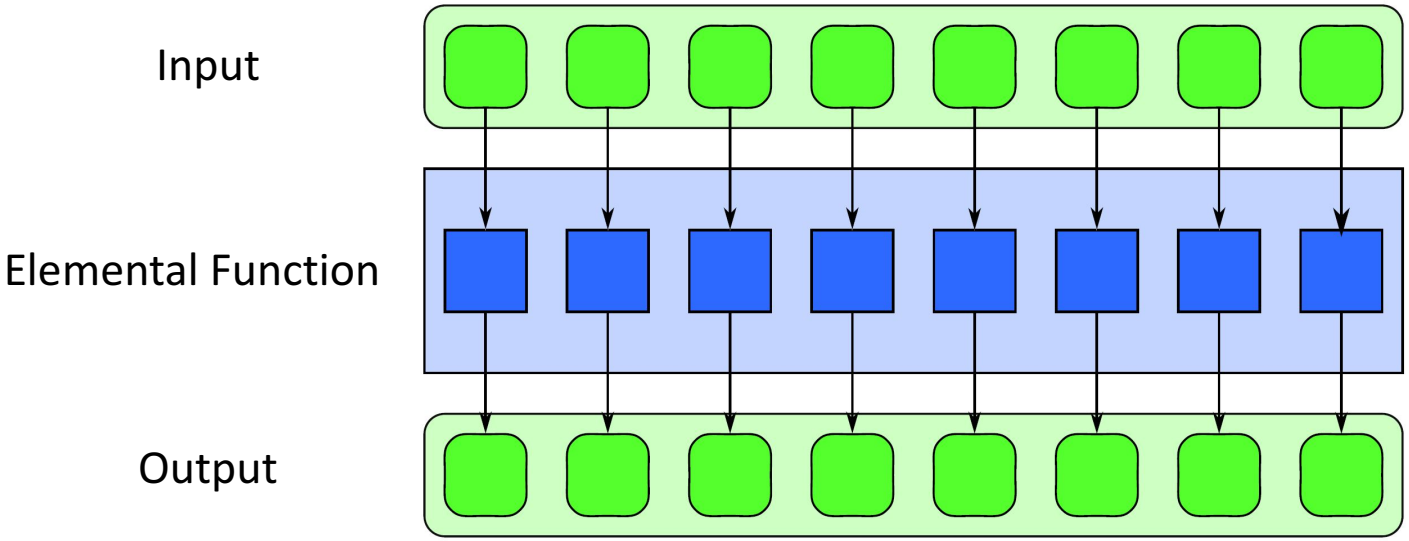
- Parallel control patterns extend serial control patterns
- Each parallel control pattern is related to at least one serial control pattern, but relaxes assumptions of serial control patterns
- Parallel control patterns: **fork-join**, **map**, **stencil**, **reduction**, **scan**, **recurrence**

Parallel Control Patterns: Fork-Join

- **Fork-join**: allows control flow to fork into multiple parallel flows, then rejoin later
- Cilk Plus implements this with **spawn** and **sync**
  - The call tree is a parallel call tree and functions are spawned instead of called
  - Functions that spawn another function call will continue to execute
  - Caller *syncs* with the spawned function to join the two
- A "join" is different than a "barrier
  - Sync – only one thread continues
  - Barrier – all threads continue

Parallel Control Patterns: Map

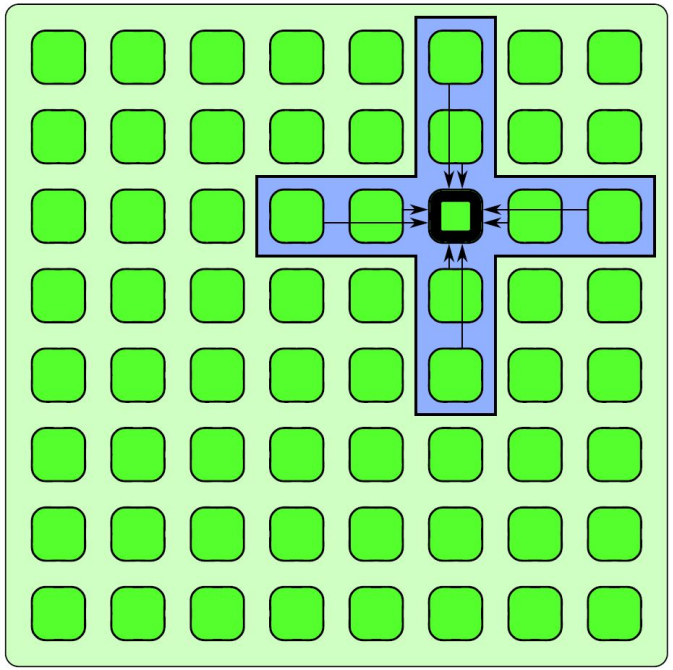
- **Map**: performs a function over every element of a collection
- Map replicates a serial iteration pattern where each iteration is independent of the others, the number of iterations is known in advance, and computation only depends on the iteration count and data from the input collection
- The replicated function is referred to as an "elemental function"



Parallel Control Patterns: Stencil

- **Stencil**: Elemental function accesses a set of "neighbors", stencil is a generalization of map
- Often combined with iteration – used with iterative solvers or to evolve a system through time

- Boundary conditions must be handled carefully in the stencil pattern
- See stencil lecture…

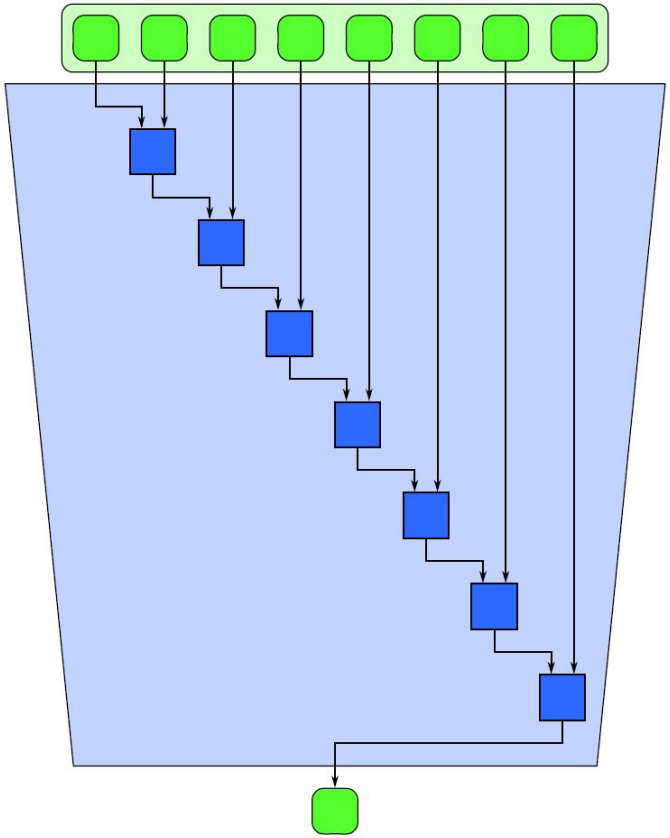


Parallel Control Patterns: Reduction

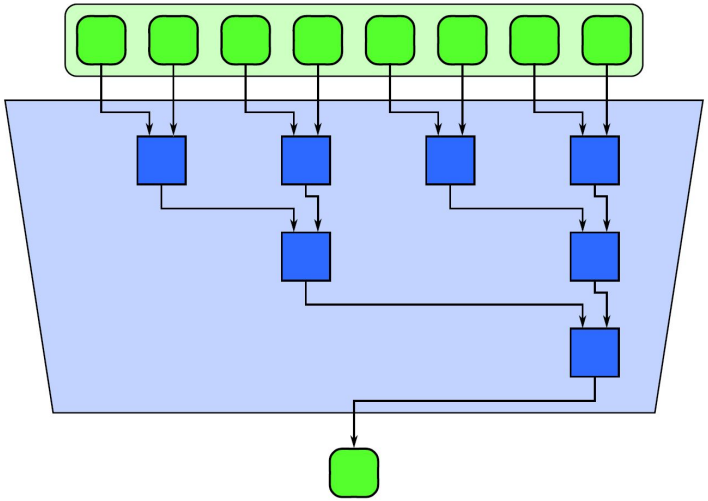
- **Reduction**: Combines every element in a collection using an associative "combiner function"
- Because of the associativity of the combiner function, different orderings of the reduction are possible
- Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR, and XOR

Parallel Control Patterns: Reduction

Serial Reduction



Parallel Reduction

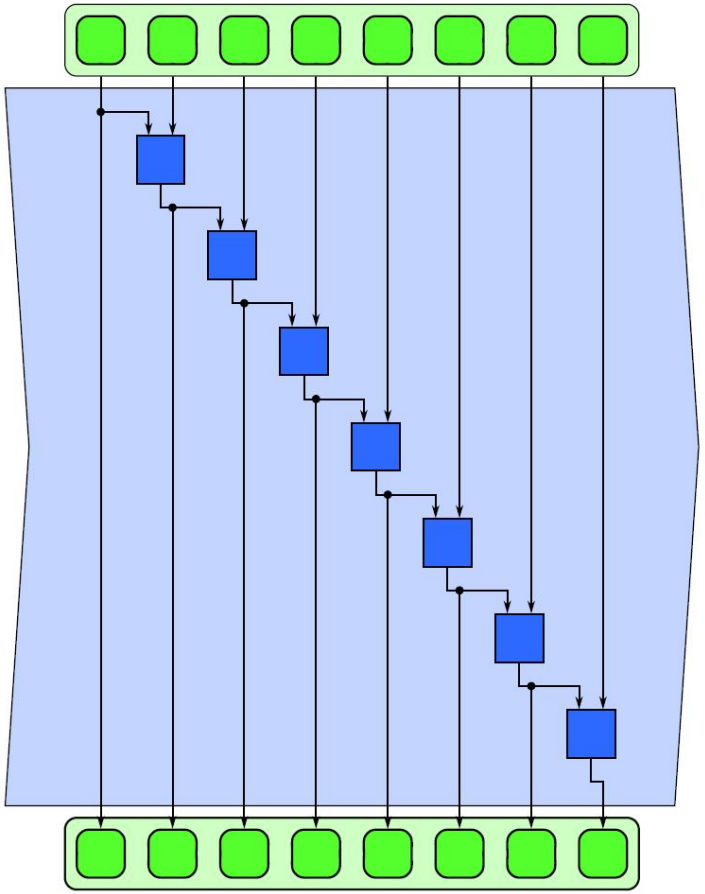


Parallel Control Patterns: Scan

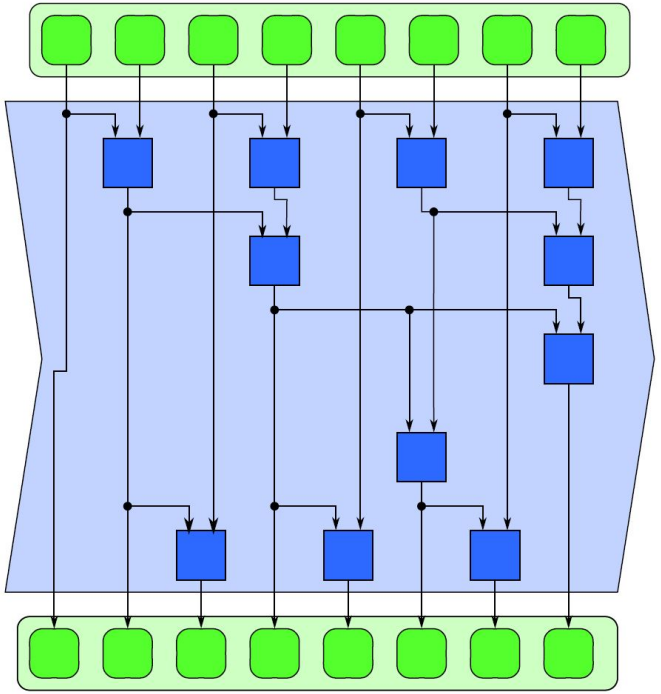
- **Scan**: computes all partial reduction of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- If the function being used is associative, the scan can be parallelized
- Parallelizing a scan is not obvious at first, because of dependencies to previous iterations in the serial loop
- A parallel scan will require more operations than a serial version

Parallel Control Patterns: Scan

Serial Scan



Parallel Scan



Parallel Control Patterns: Recurrence

- **Recurrence**: More complex version of map, where the loop iterations can depend on one another
- Similar to map, but elements can use outputs of adjacent elements as inputs
- For a recurrence to be computable, there *must* be a serial ordering of the recurrence elements so that elements can be computed using previously computed outputs

Serial Data Management Patterns

- Serial programs can manage data in many ways
- Data management deals with how data is allocated, shared, read, written, and copied
- Serial Data Management Patterns: **random read and write**, **stack allocation**, **heap allocation**, **objects**

Serial Data Management Patterns: random read and write

- Memory locations indexed with addresses
- Pointers are typically used to refer to memory addresses
- Aliasing (uncertainty of two pointers referring to the same object) can cause problems when serial code is parallelized

Serial Data Management Patterns: Stack Allocation

- Stack allocation is useful for dynamically allocating data in LIFO manner
- Efficient – arbitrary amount of data can be allocated in constant time
- Stack allocation also preserves locality
- When parallelized, typically each thread will get its own stack so thread locality is preserved

Serial Data Management Patterns: Heap Allocation

- Heap allocation is useful when data cannot be allocated in a LIFO fashion
- But, heap allocation is slower and more complex than stack allocation
- A parallelized heap allocator should be used when dynamically allocating memory in parallel
  - This type of allocator will keep separate pools for each parallel worker

Serial Data Management Patterns: Objects

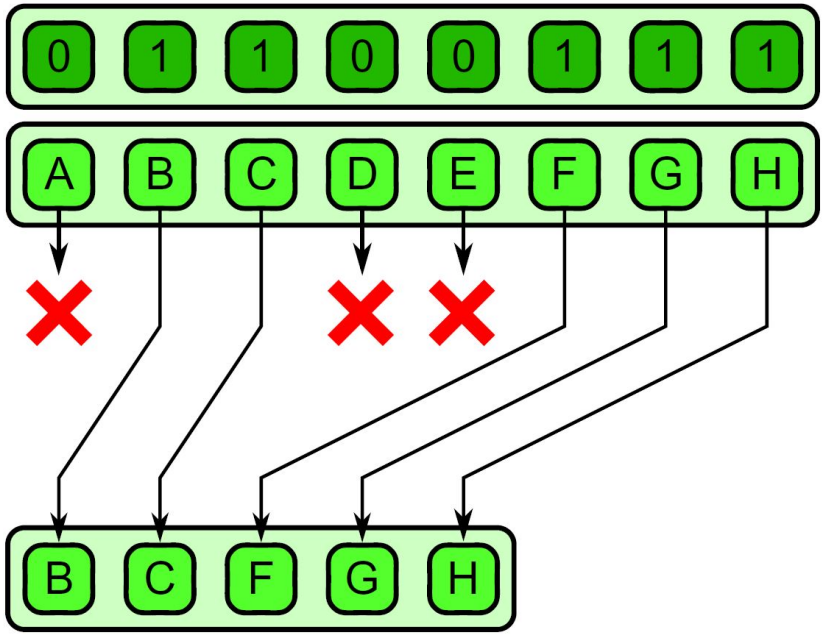
- Objects are language constructs to associate data with code to manipulate and manage that data
- Objects can have member functions, and they also are considered members of a class of objects
- Parallel programming models will generalize objects in various ways

Parallel Data Management Patterns

- To avoid things like race conditions, it is critically important to know when data is, and isn't, potentially shared by multiple parallel workers
- Some parallel data management patterns help us with data locality
- Parallel data management patterns: **pack**, **pipeline**, **geometric decomposition**, **gather**, and **scatter**

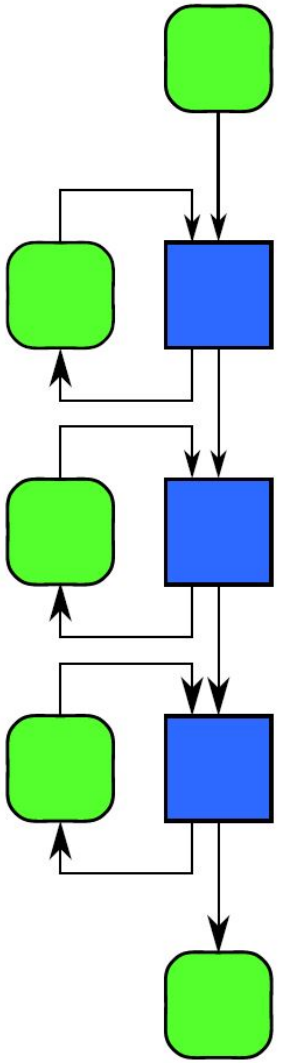
Parallel Data Management Patterns: Pack

- **Pack** is used eliminate unused space in a collection
- Elements marked *false* are discarded, the remaining elements are placed in a contiguous sequence in the same order
- Useful when used with map
- **Unpack** is the inverse and is used to place elements back in their original locations



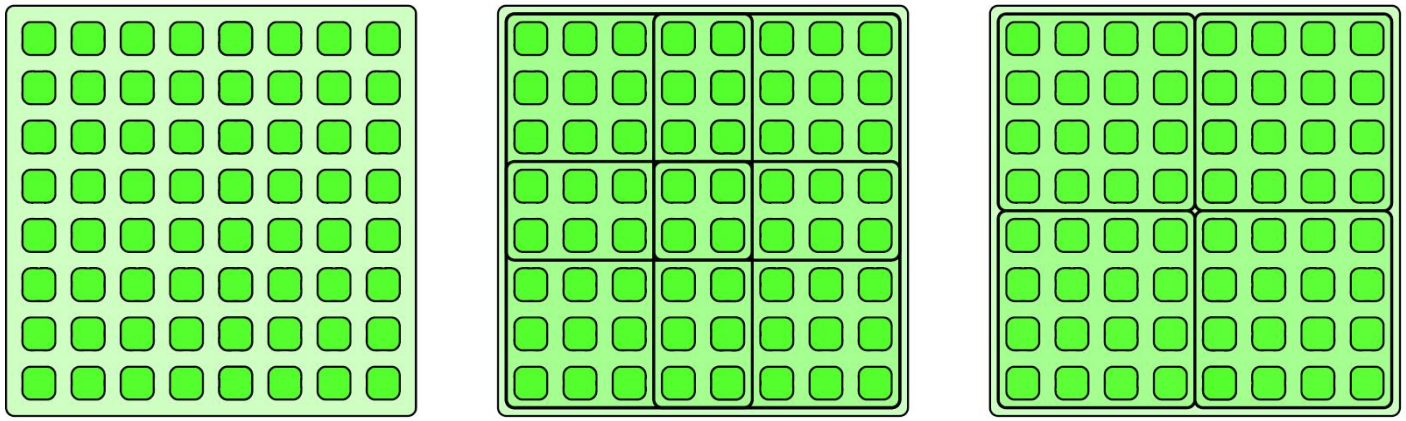
Parallel Data Management Patterns: Pipeline

- **Pipeline** connects tasks in a producer-consumer manner
- A linear pipeline is the basic pattern idea, but a pipeline in a DAG is also possible
- Pipelines are most useful when used with other patterns as they can multiply available parallelism



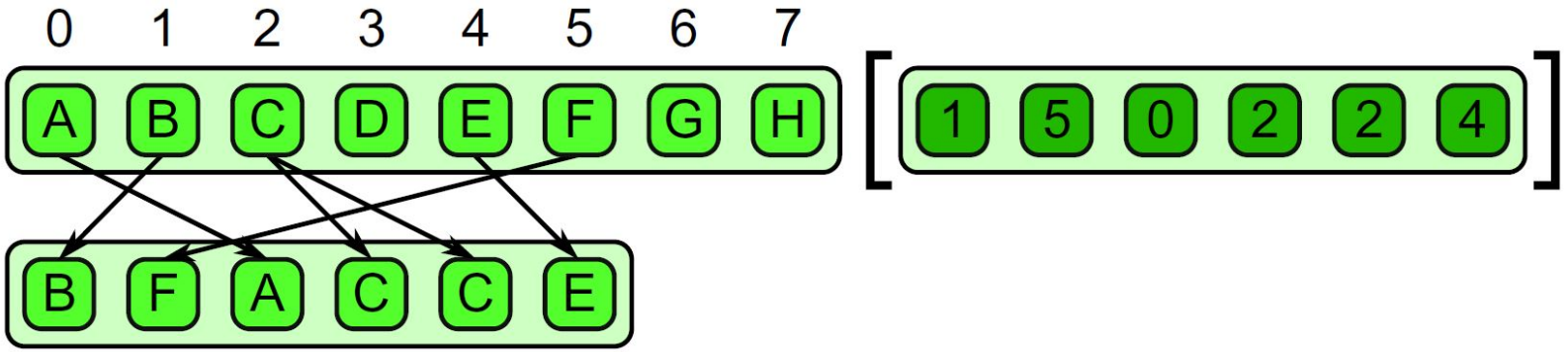
Parallel Data Management Patterns: Geometric Decomposition

- **Geometric Decomposition** – arranges data into subcollections
- Overlapping and non-overlapping decompositions are possible
- This pattern doesn't necessarily move data, it just gives us another view of it



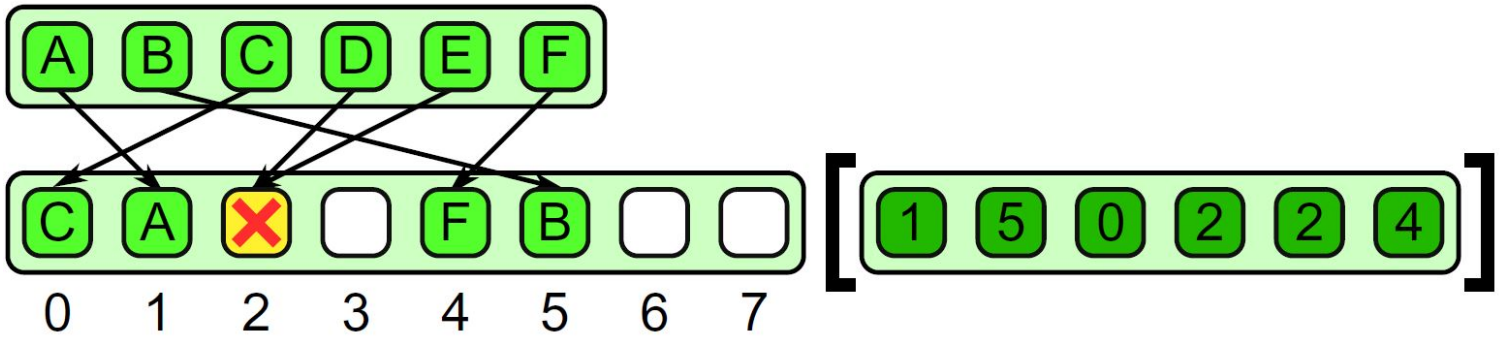
Parallel Data Management Patterns: Gather

- **Gather** reads a collection of data given a collection of indices
- Think of a combination of map and random serial reads
- The output collection shares the same type as the input collection, but it share the same shape as the indices collection



Parallel Data Management Patterns: Scatter

- **Scatter** is the inverse of gather
- A set of input and indices is required, but each element of the input is written to the output at the given index instead of read from the input at the given index
- Race conditions can occur when we have two writes to the same location!



Other Parallel Patterns

- **Superscalar Sequences**: write a sequence of tasks, ordered only by dependencies
- **Futures**: similar to fork-join, but tasks do not need to be nested hierarchically
- **Speculative Selection**: general version of serial selection where the condition and both outcomes can all run in parallel
- **Workpile**: general map pattern where each instance of elemental function can generate more instances, adding to the "pile" of work

Other Parallel Patterns

- **Search**: finds some data in a collection that meets some criteria
- **Segmentation**: operations on subdivided, non-overlapping, non-uniformly sized partitions of 1D collections
- **Expand**: a combination of pack and map
- **Category Reduction**: Given a collection of elements each with a label, find all elements with same label and reduce them

Programming Model Support for Patterns

Table 3.1 Summary of programming model support for the serial patterns discussed in this book. Note that some of the parallel programming models we consider do not, in fact, support all the common serial programming patterns. In particular, note that recursion and memory allocation are limited on some model.

Serial Pattern	TBB	Cilk Plus	OpenMP	ArBB	OpenCL
(Serial) Nesting	F	F	F	F	F
Sequence	F	F	F	F	F
Selection	F	F	F	F	F
Iteration	F	F	F	F	F
Recursion	F	F	F		?
Random Read	F	F	F	F	F
Random Write	F	F	F		F
Stack Allocation	F	F	F		?
Heap Allocation	F	F	F		
Closures				F	F
Objects	F	F	F(w/C++)	F	

Programming Model Support for Patterns

Table 3.2 Summary of programming model support for the patterns discussed in this book. F: Supported directly, with a special feature. I: Can be implemented easily and efficiently using other features. P: Implementations of one pattern in terms of others, listed under the pattern being implemented. Blank means the particular pattern cannot be implemented in that programming model (or that an efficient implementation cannot be implemented easily). When examples exist in this book of a particular pattern with a particular model, section references are given.

Parallel Pattern	TBB	Cilk Plus	OpenMP	ArBB	OpenCL
Parallel nesting	F	F			
Map	F 4.2.3; 4.3.3 11	F 4.2.4;4.2.5; 4.3.4;4.3.5 11	F 4.2.6; 4.3.6	F 4.2.7;4.2.8; 4.3.7	F 4.2.9; 4.3.8
Stencil	I 10	I 10	I	F 10	I
Workpile	F				I
Reduction	F 5.3.4 11	F 5.3.5 11	F 5.3.6	F 5.3.7	I
Scan	F 5.6.5 14	I 5.6.3 *P 8.11* 14	I 5.6.4 *P 5.4.4*	F 5.6.6	I
Fork–join	F 8.9.2 13	F 8.7; 8.9.1 13	I		
Recurrence		*P 8.12*			
Superscalar sequence					F
Futures					
Speculative selection					
Pack	I 14	I 14	I	F	I
Expand	I	I	I	I	I
Pipeline	F 12	I 12	I		
Geometric decomposition	I 15	I 15	I	I	I
Search	I	I	I	I	I
Category reduction	I	I	I	I	I
Gather	I	F	I	F	I
Atomic scatter	F	I	I		I
Permutation scatter	F	F	F	F	F
Merge scatter	I	I	I	F	I
Priority scatter					

Programming Model Support for Patterns

Table 3.3 Additional patterns discussed. F: Supported directly, with a special feature. I: Can be implemented easily and efficiently using other features. Blank means the particular pattern cannot be implemented in that programming model (or that an efficient implementation cannot be implemented easily).

Parallel Pattern	TBB	Cilk Plus	OpenMP	ArBB	OpenCL
Superscalar sequence	I	I	I		F
Futures	I	I	I		I
Speculative selection	I				
Workpile	F	I	I		I
Expand	I	I	I	I	I
Search	I	I	I	I	I
Category reduction	I	I	I	I	I
Atomic scatter	F	I	I		I
Permutation scatter	F	F	F	F	F
Merge scatter	I	I	I	F	I
Priority scatter					

Map Pattern - Overview

- Map
- Optimizations
  - Sequences of Maps
  - Code Fusion
  - Cache Fusion
- Related Patterns
- Example Implementation: Scaled Vector Addition (SAXPY)
  - Problem Description
  - Various Implementations

Mapping

- "Do the same thing many times"

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

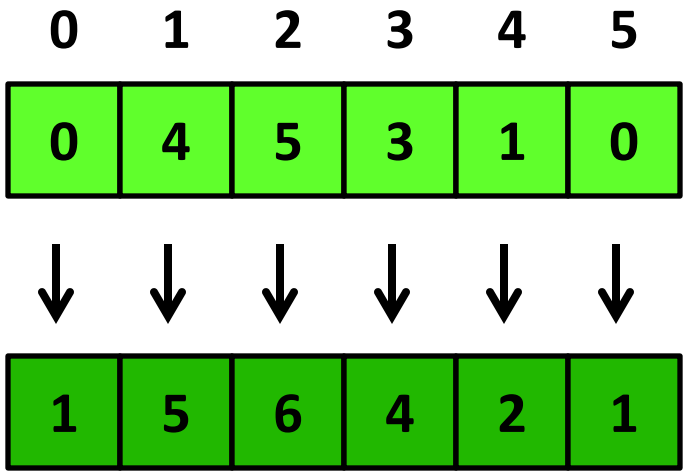
- Well-known higher order function in languages like ML, Haskell, Scala

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

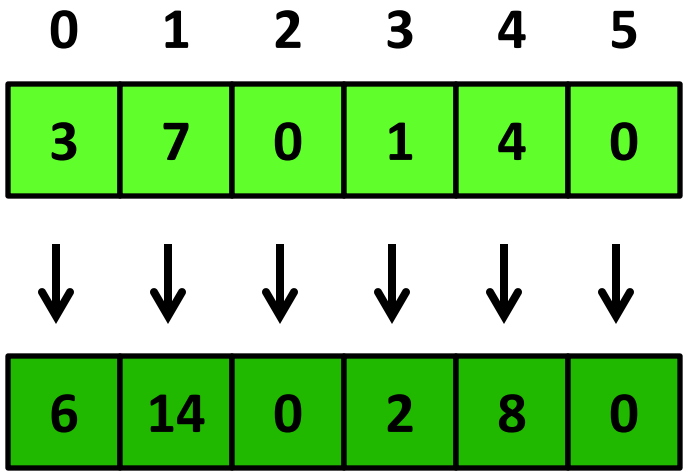
applies a function each element in a list and returns a list of results

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

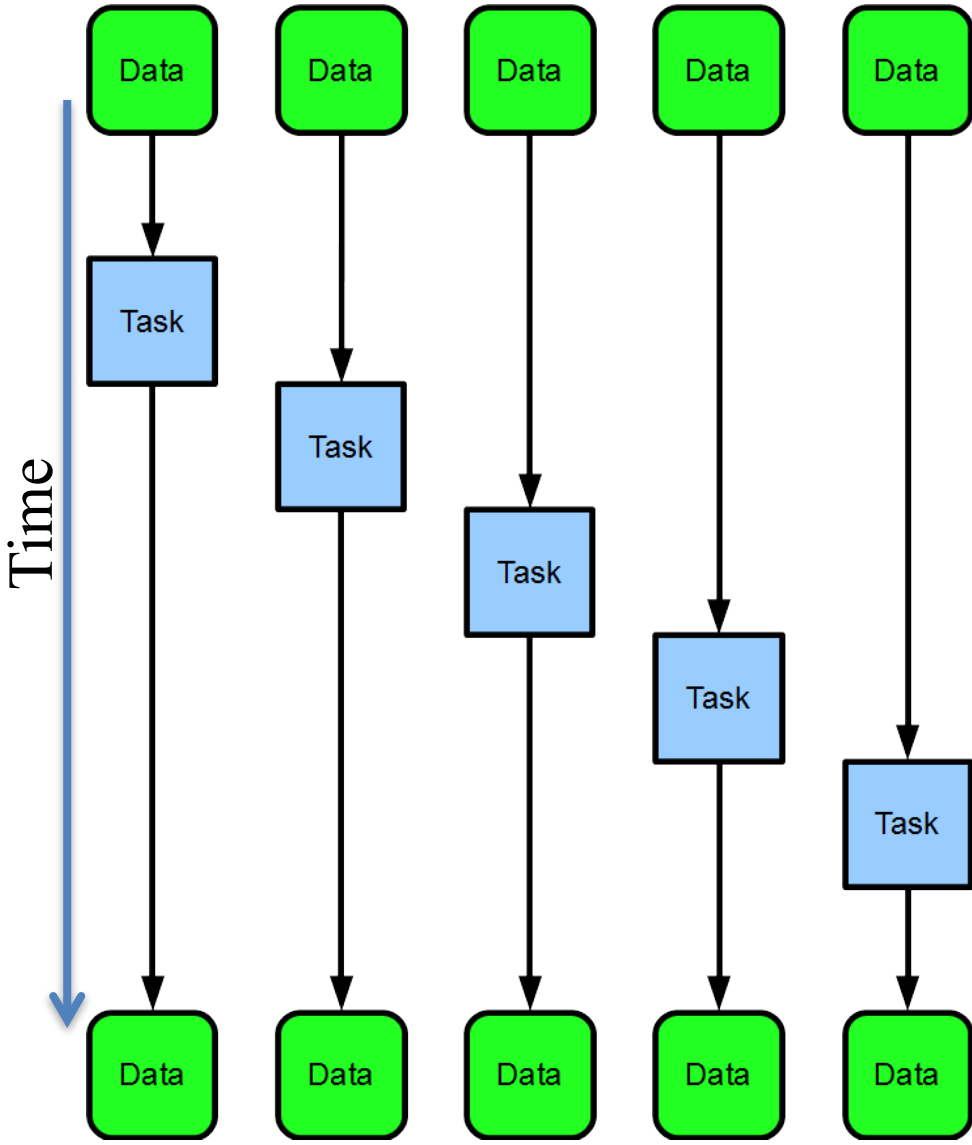
- Map is a "foreach loop" where each iteration is independent

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 $T(\infty) \in O(\log n)$.

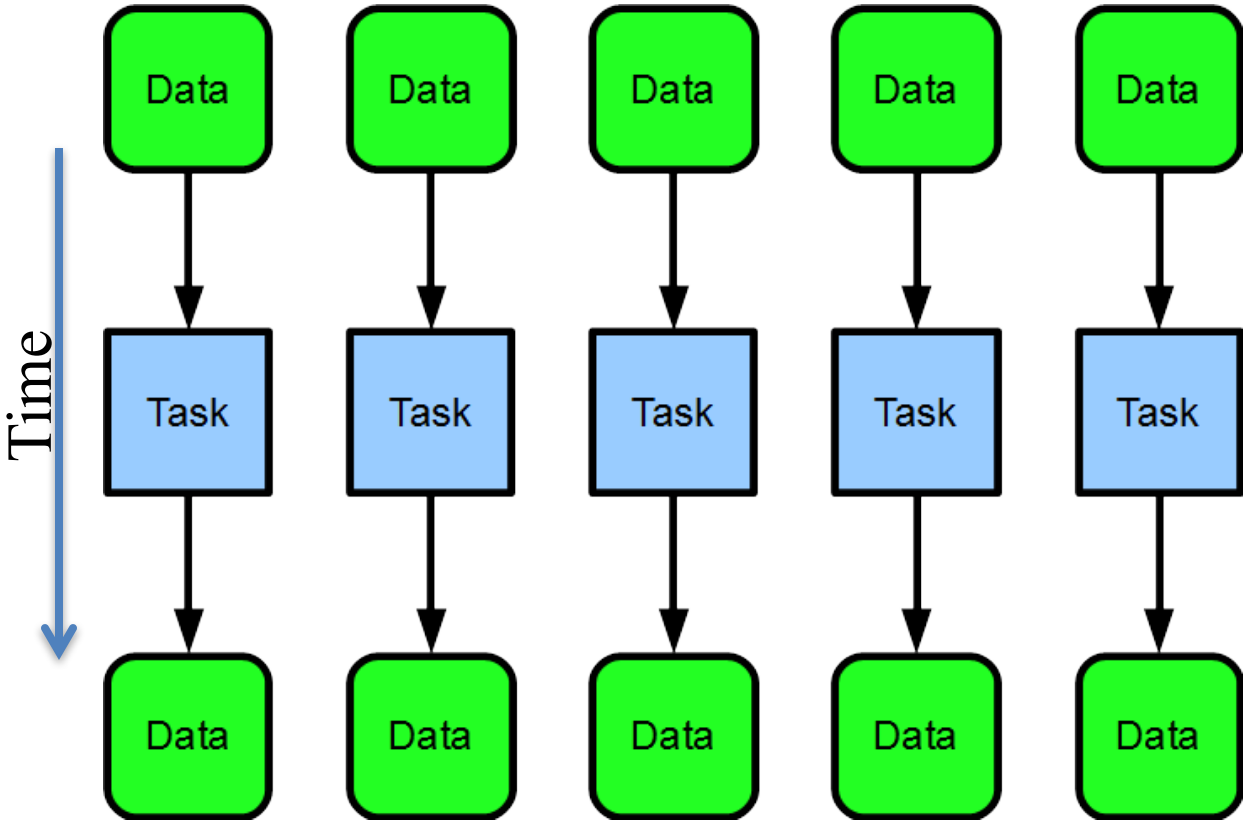
Sequential Map

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

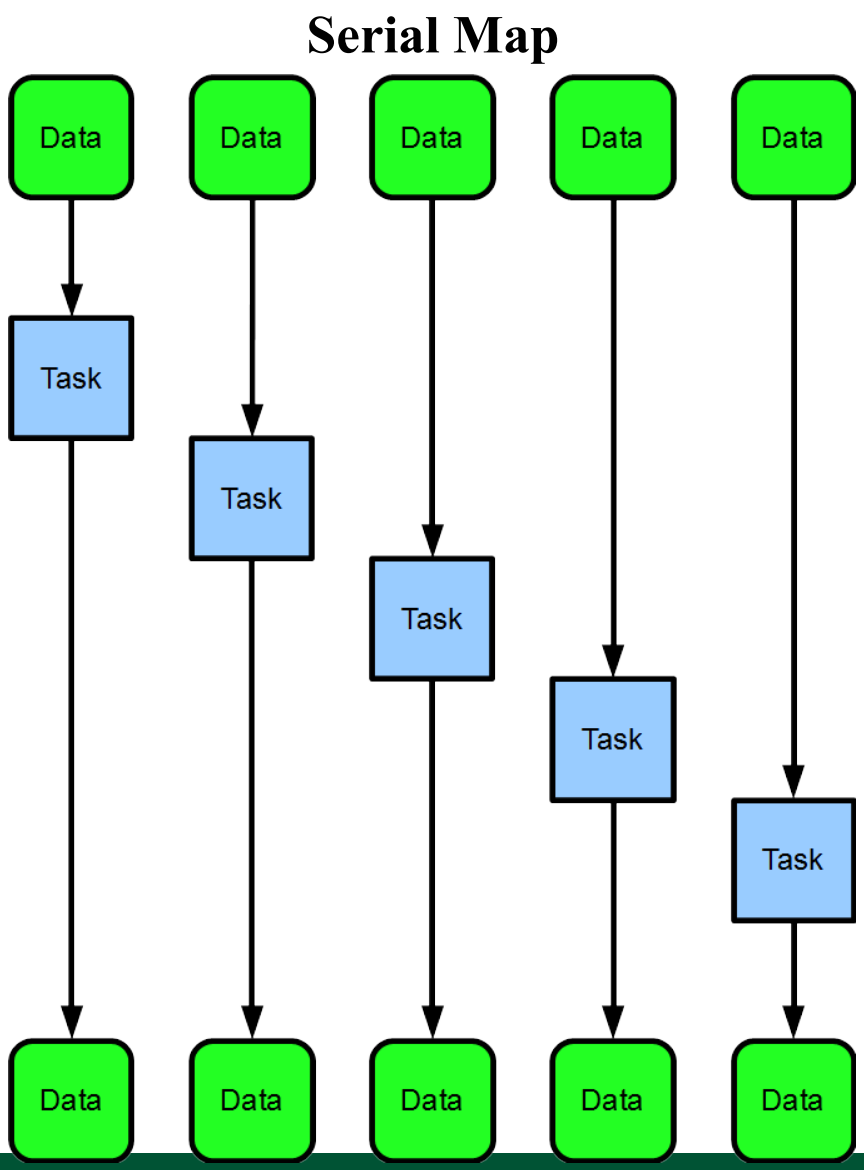


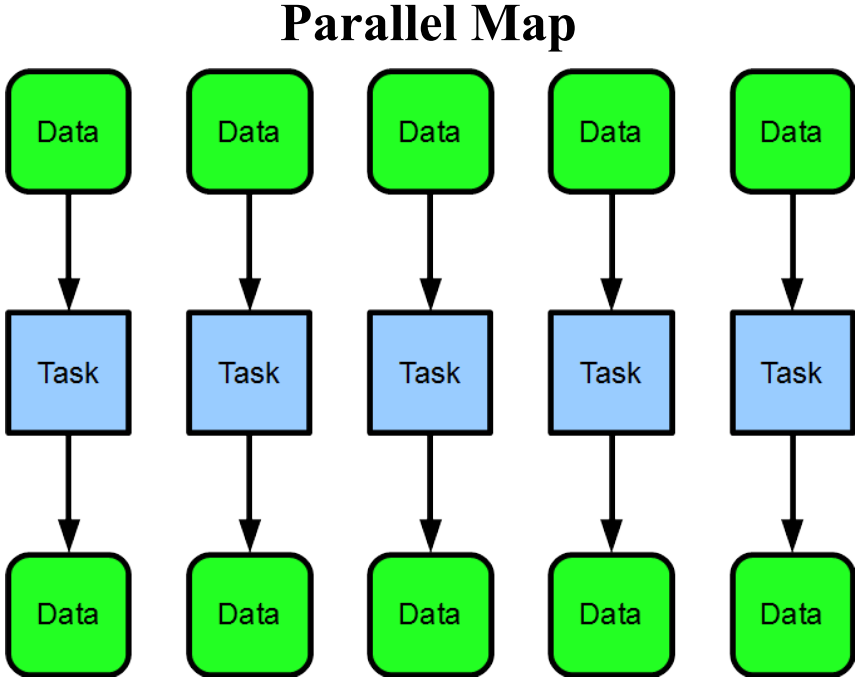
Parallel Map

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

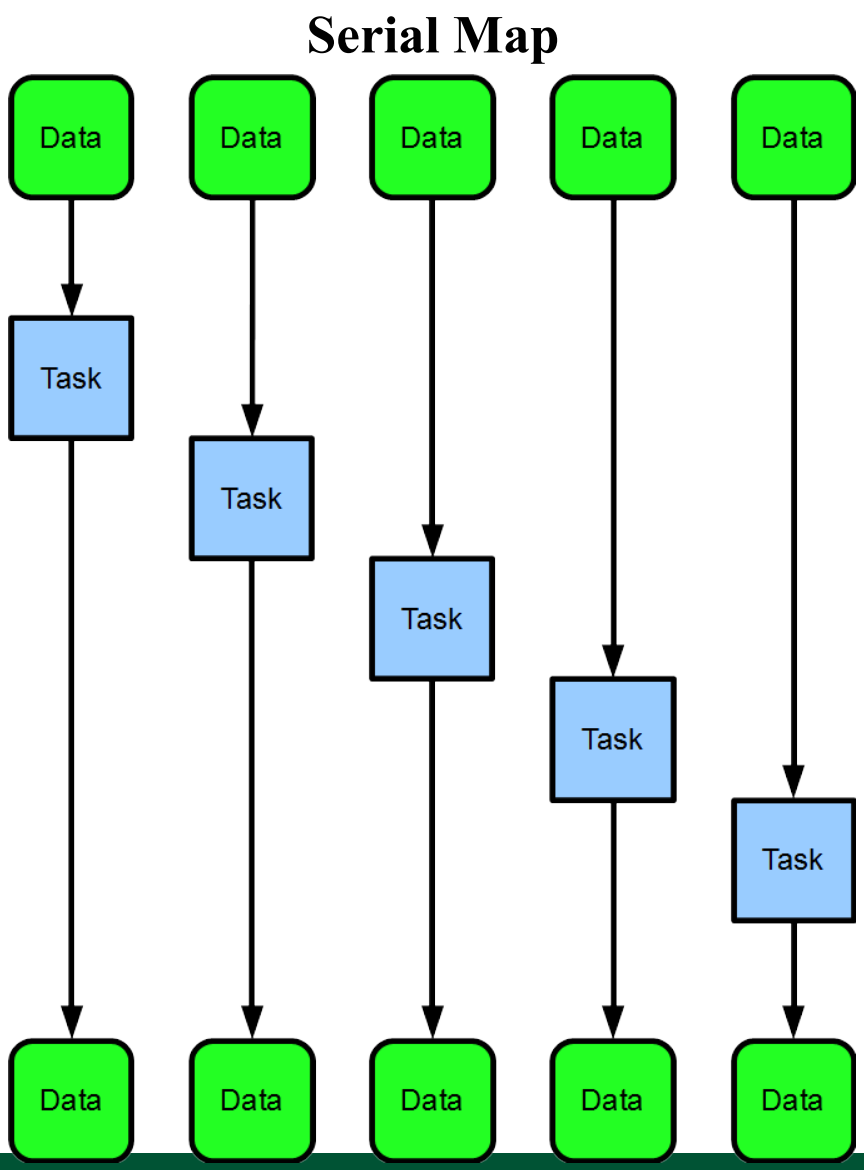


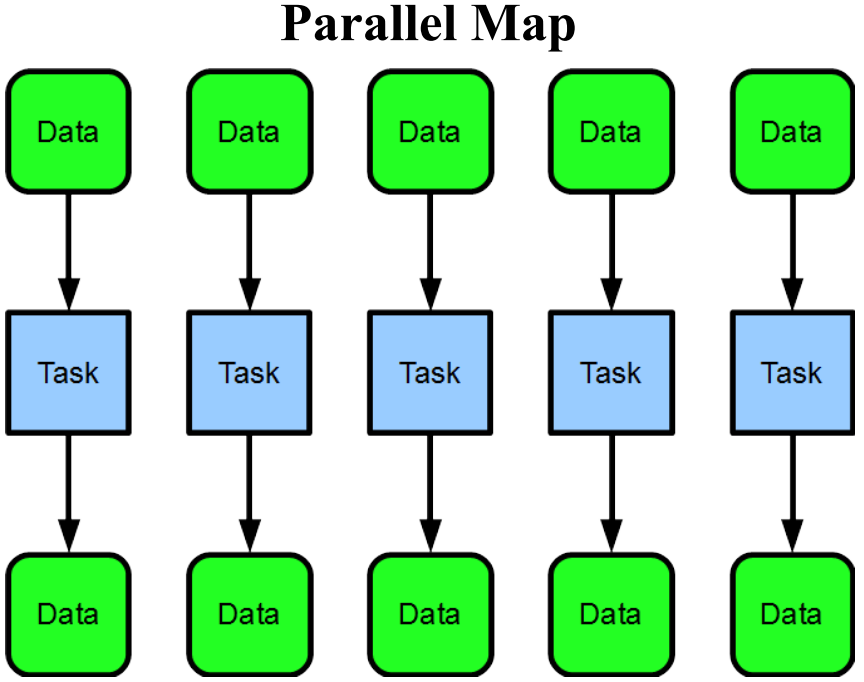
Comparing Maps





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

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;
```

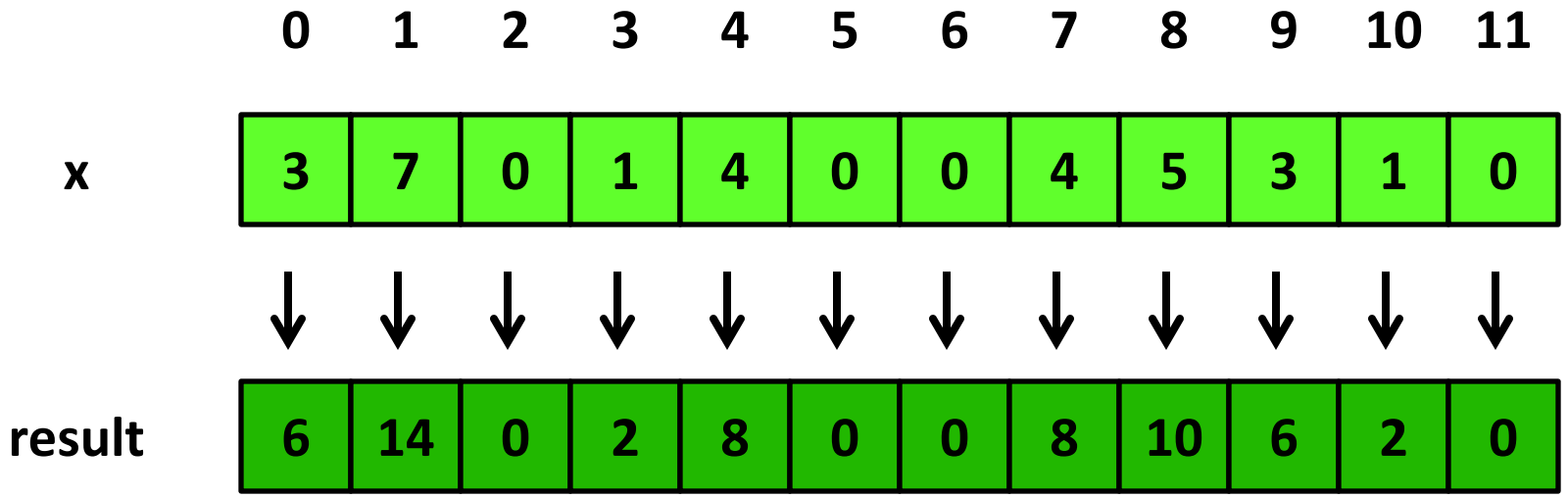
is CilkPlus array notation for "multiply every element in *A* by 5"

Unary Maps

Unary Maps

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

Map with 1 Input, 1 Output



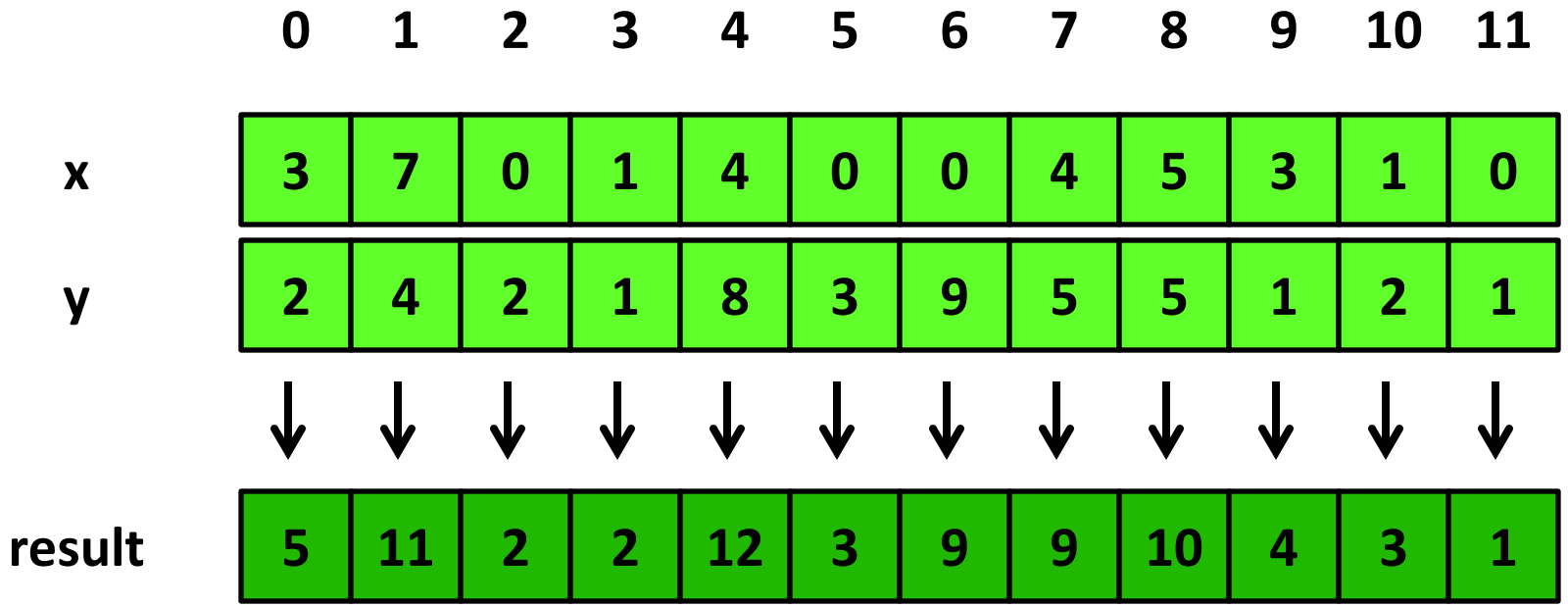

```
int  oneToOne ( int x[11] ) {
        return x*2;
}
```

N-ary Maps

N-ary Maps

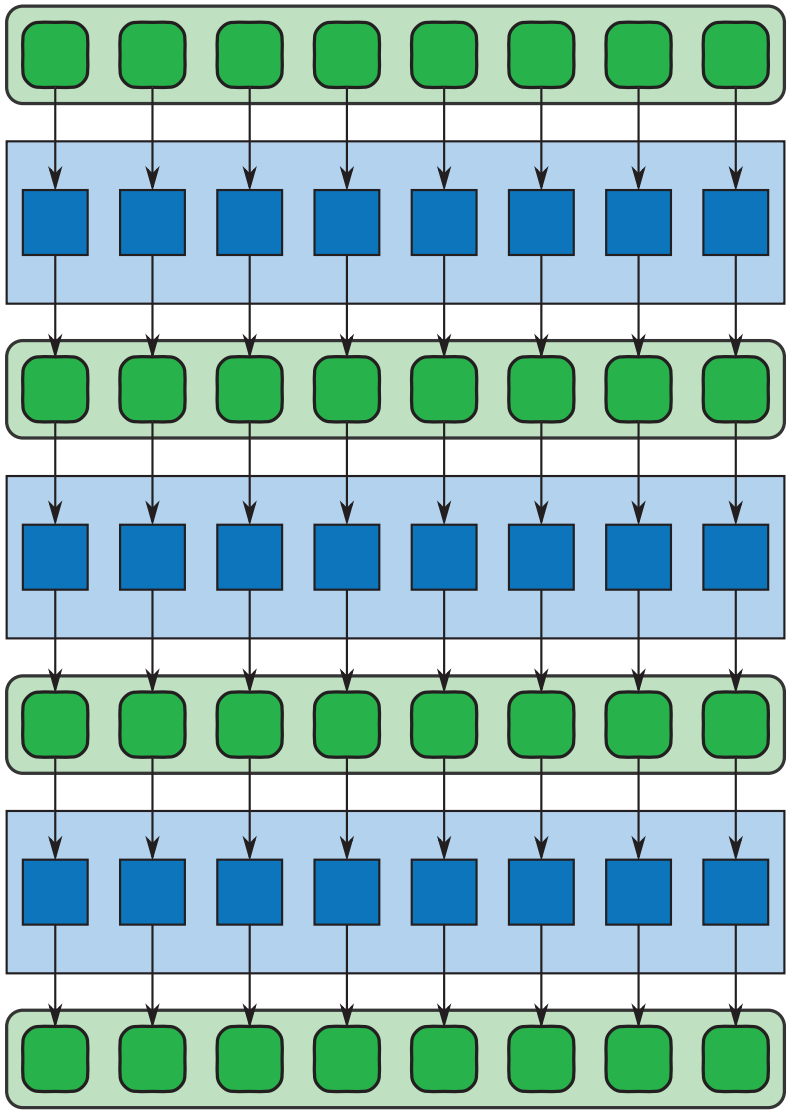
But, sometimes it makes sense to map over multiple collections at once…

Map with 2 Inputs, 1 Output



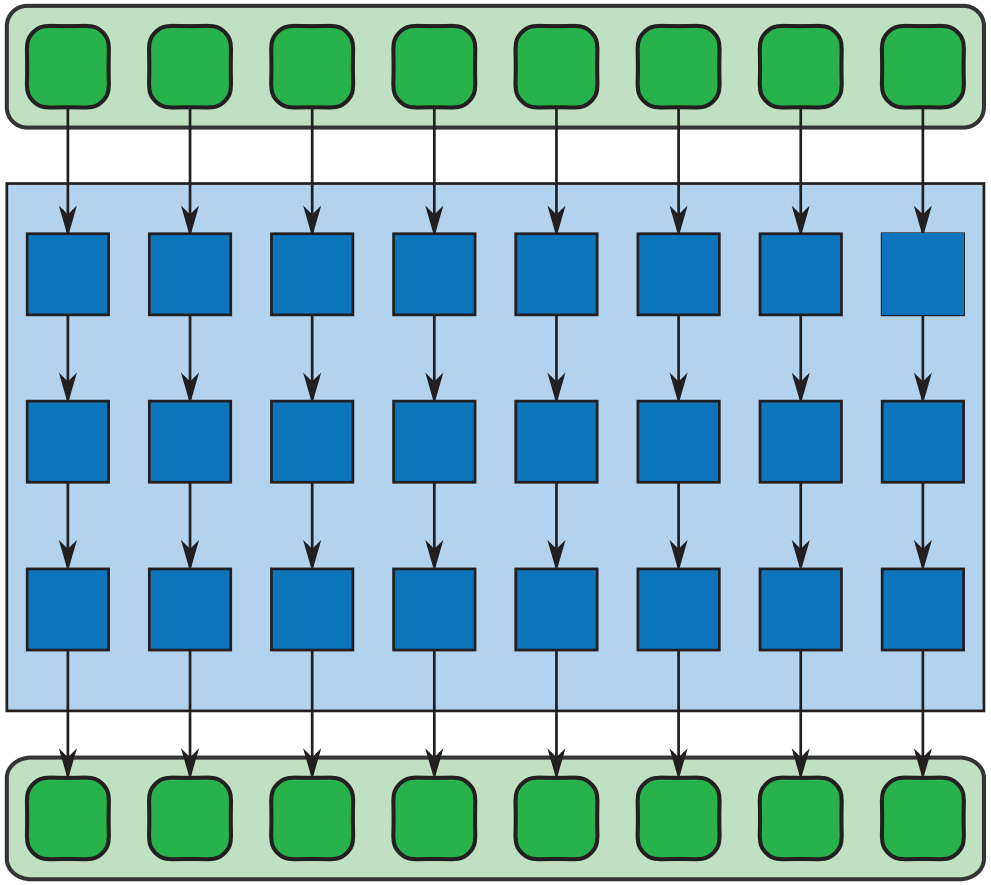

```
int  twoToOne ( int x[11], int y[11] ) {
        return x+y;
}
```

Optimization – Sequences of Maps



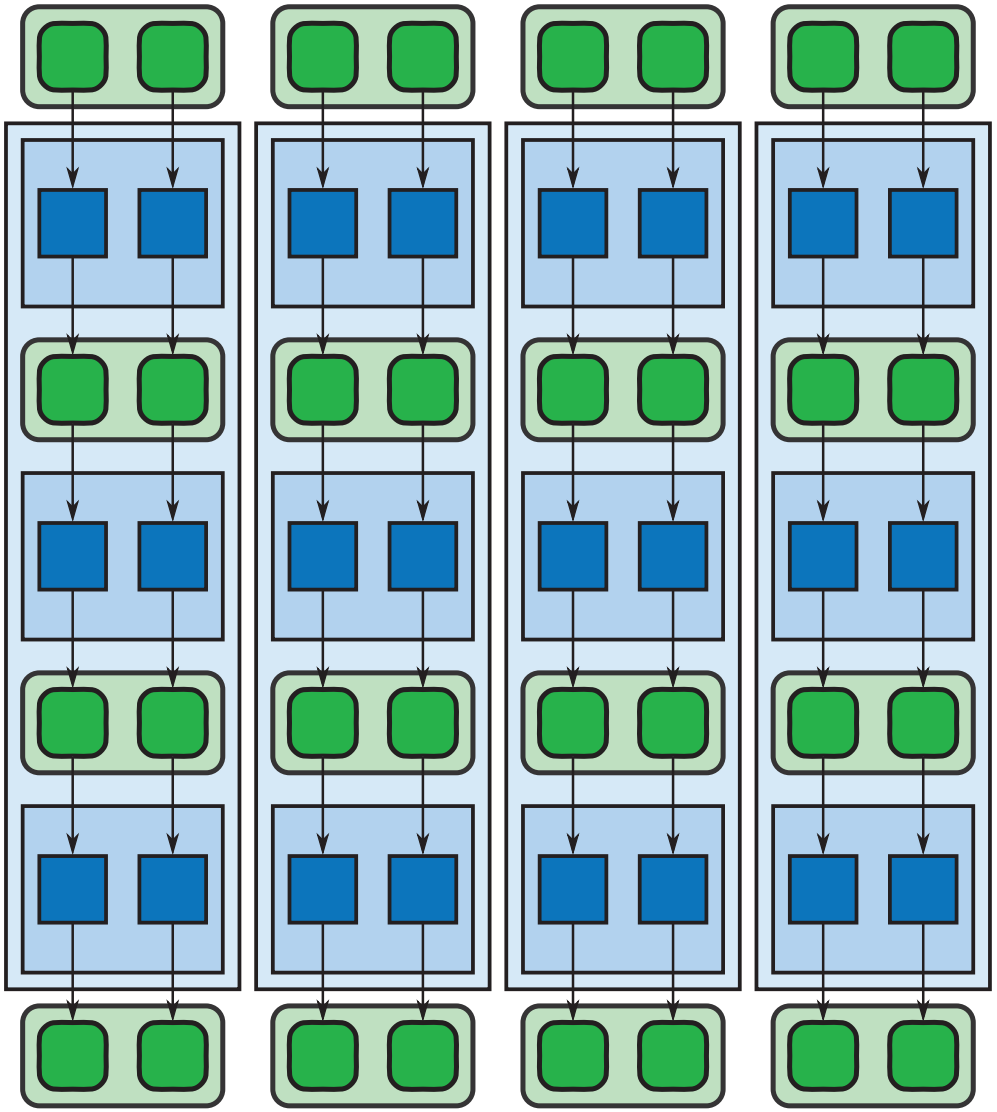
- Often several map operations occur in sequence
  - Vector math consists of many small operations such as additions and multiplications applied as maps
- A naïve implementation may write each intermediate result to memory, wasting memory BW and likely overwhelming the cache

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

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

Related Patterns

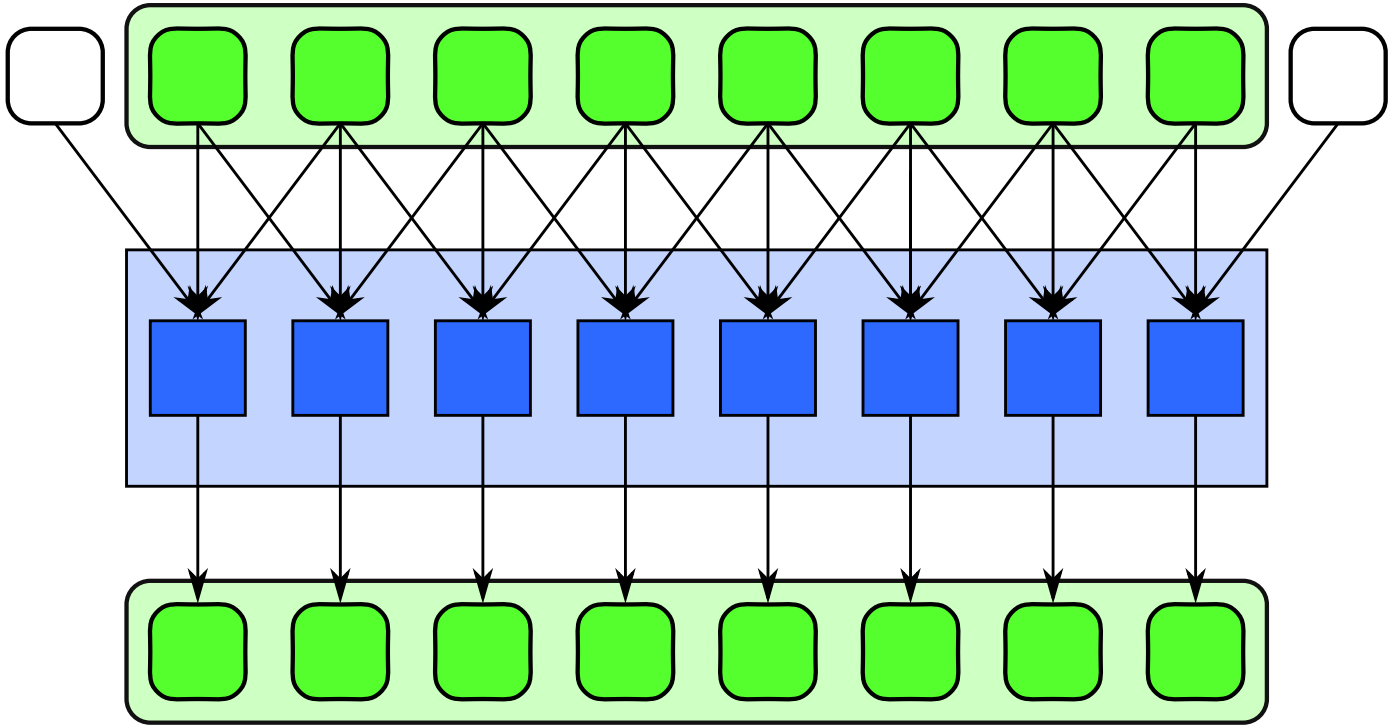
Three patterns related to map are discussed here:

- Stencil
- Workpile
- Divide-and-Conquer

More detail presented in a later lecture

Stencil

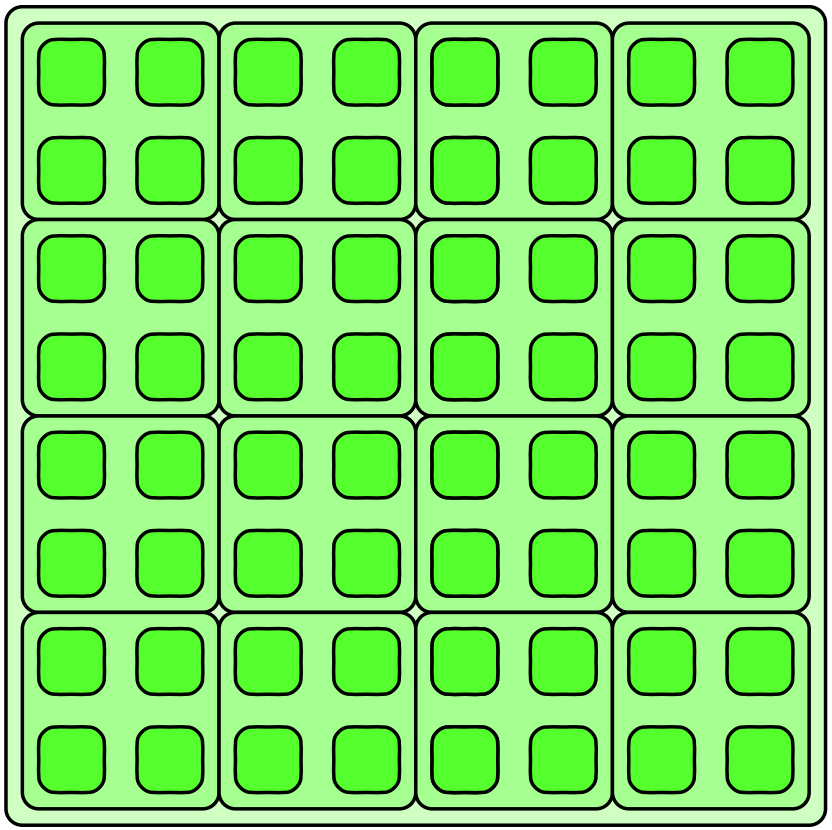
- Each instance of the map function accesses neighbors of its input, offset from its usual input
- Common in imaging and PDE solvers



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

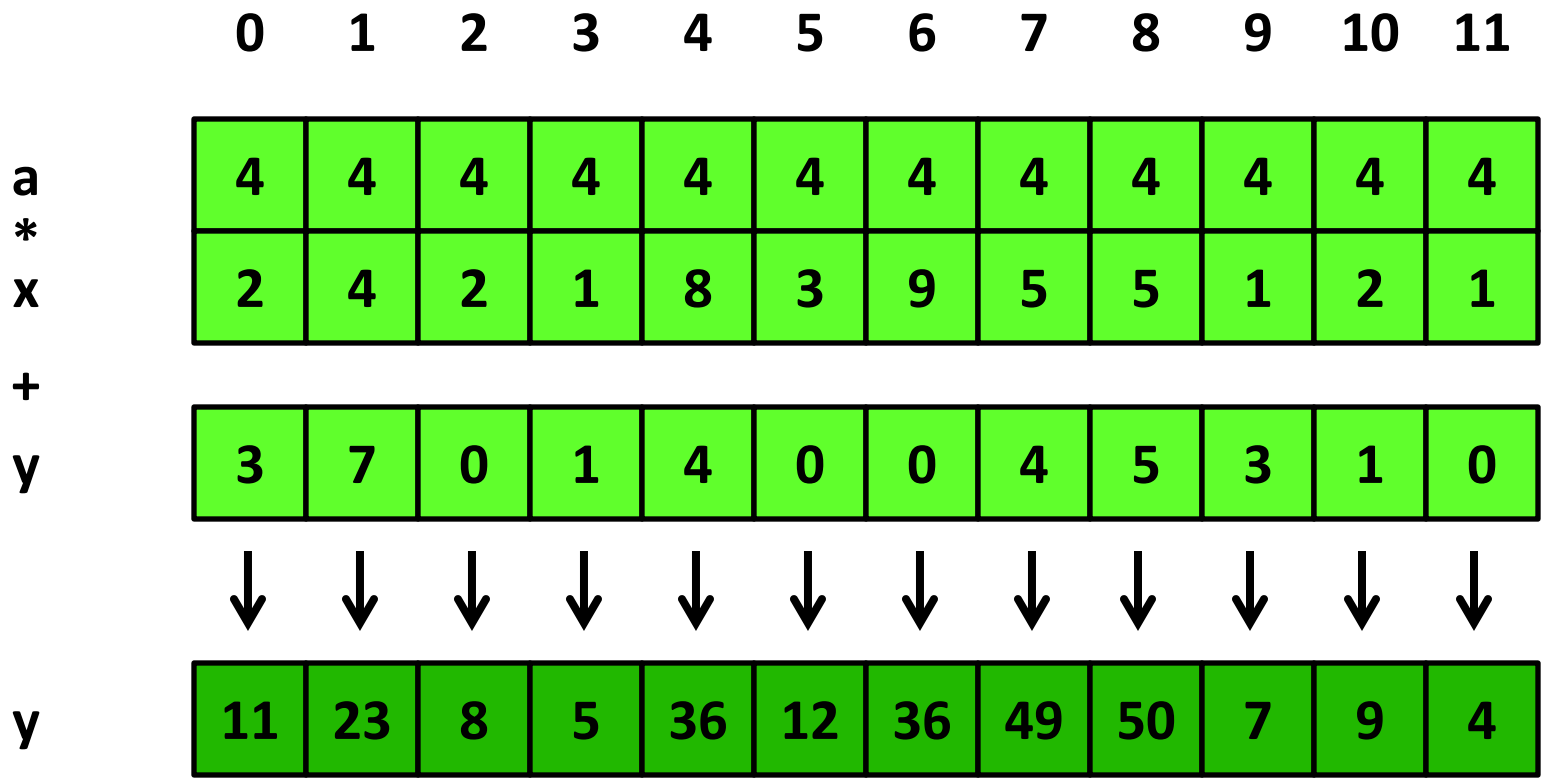


- Applies if a problem can be divided into smaller subproblems recursively until a base case is reached that can be solved serially

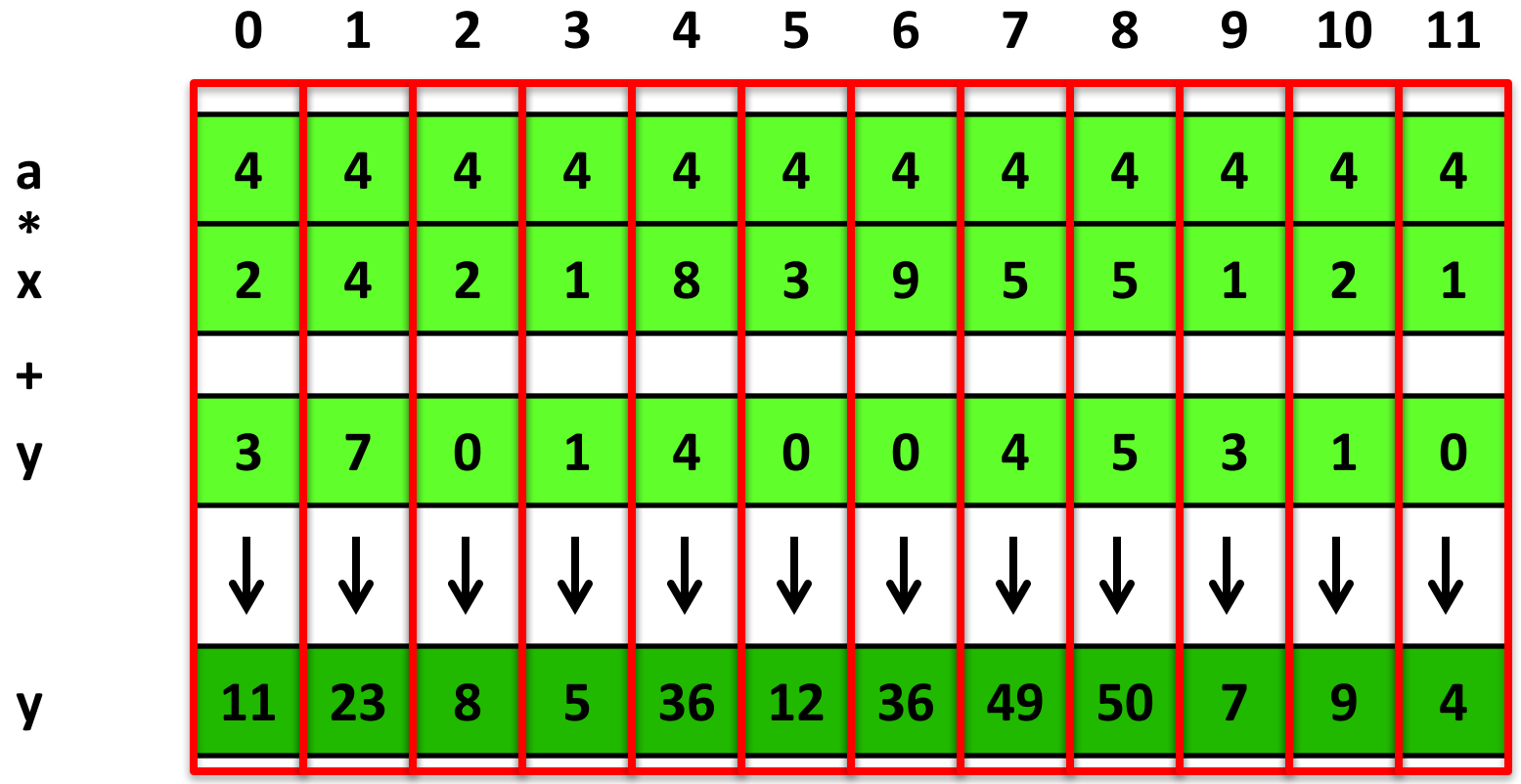
Example: Scaled Vector Addition (SAXPY)

- $y \leftarrow ax + y$
  - 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?

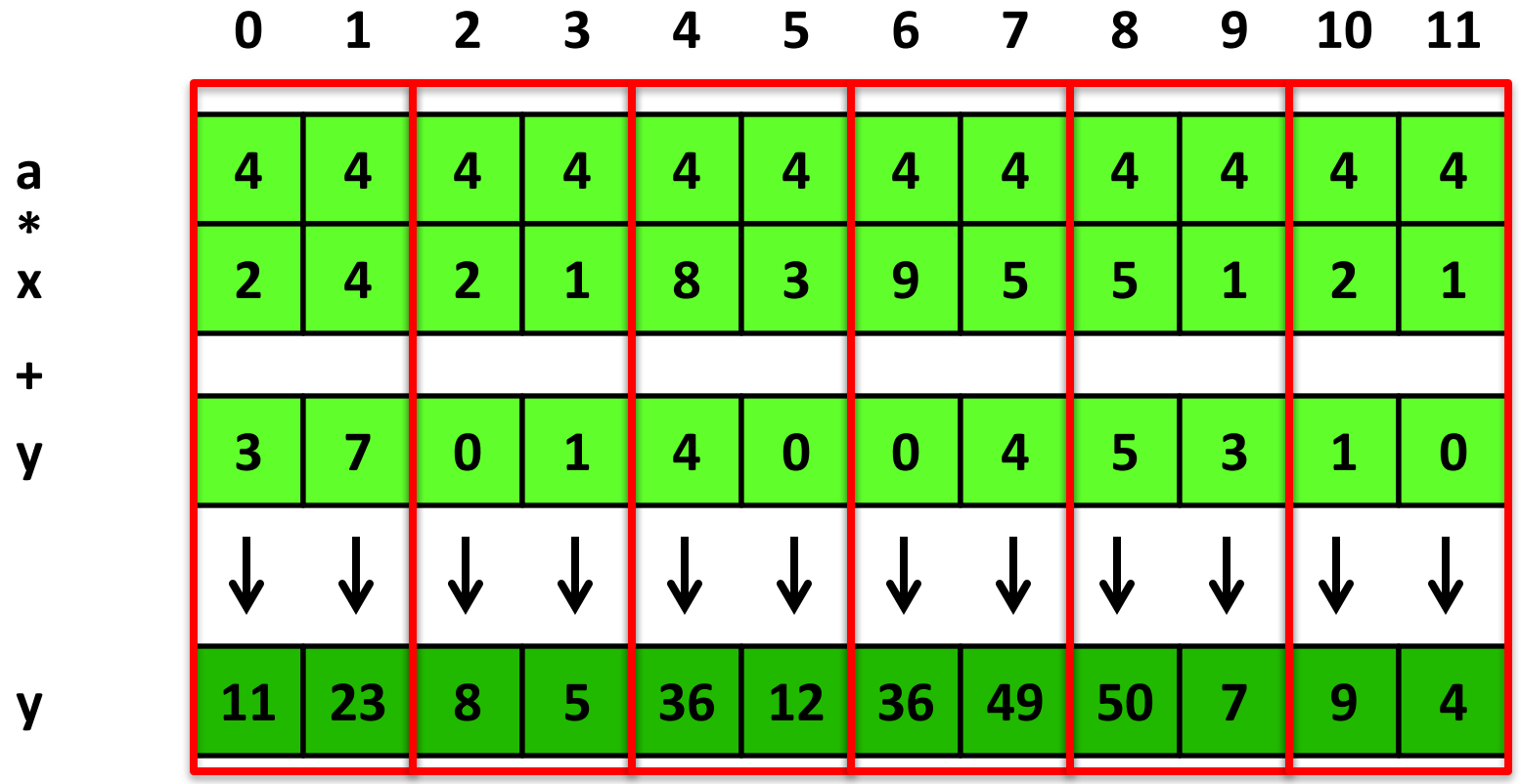


Visual: $y \leftarrow ax + y$



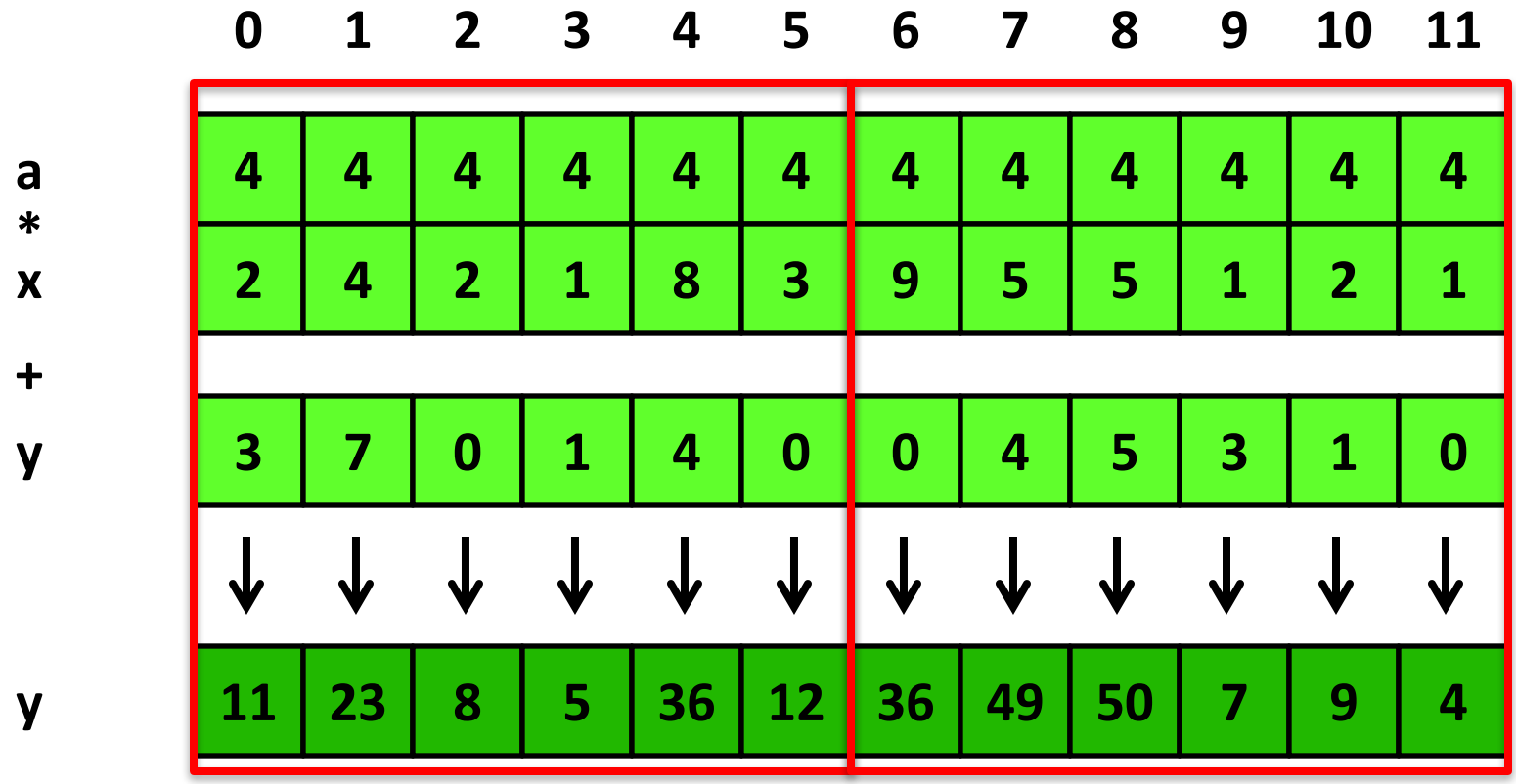
Twelve processors used → one for each element in the vector

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  }
```

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 }
```

OpenMP SAXPY Performance

Vector size = 500,000,000