Parallel Programming Patterns

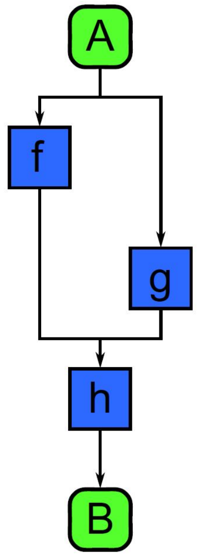
Bryan Mills, PhD

Spring 2017

What is a programming patterns?

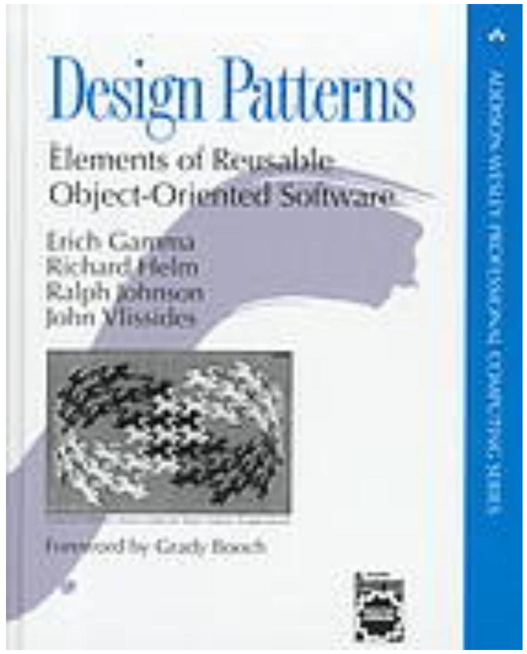
- Repeatable solution to commonly occurring problem
- It isn’t a solution that you can’t simply apply, the engineer has to implement it
- Why?
  - Familiar & Understood
  - Tested & Optimized

```
T = f(A);
S = g(A);
B = h(S,T);
```



History of Patterns?

- Patterns date back to late 1970s but…
- Became really popular with the “Gang of Four” book
  - This book described object oriented patterns
  - Published in 1994
- Famous patterns: (as defined by me)
  - Decorator
  - Observer
  - Adaptor
  - Factory
  - Singleton
  - Prototype
  - Iterator


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Why Patterns?

- Patterns help "structure" your code
- Sharing code becomes easier
  - "I'm using a singleton here…"
- Patterns have been "proven" and are understood so if you follow one you benefit from common knowledge
- Languages often "support" a particular pattern
  - Ex: decorator, map, reducer
- Optimization can be done without the programmers knowledge
  - Ex: parallelize a map call, copy of modify factory

Serial Patterns

The following patterns are the basis of “structured programming” for serial computation:

- Sequence
- Selection
- Iteration
- Nesting
- Functions
- Recursion
- Random read
- Random write
- Stack allocation
- Heap allocation
- Objects
- Closures

Compositions of these patterns can be used in place of unstructured mechanisms such as “goto.”

Parallel Patterns

The following additional parallel patterns can be used for "structured parallel programming":

- Superscalar sequence
- Speculative selection
- Map
- Recurrence
- Scan
- Reduce
- Pack/expand
- Fork/join
- Pipeline
- Partition
- Segmentation
- Stencil
- Search/match
- Gather
- Merge scatter
- Priority scatter
- Permutation scatter
- Atomic scatter

Using these patterns, threads and vector intrinsic can (mostly) be eliminated and the maintainability of software improved.

Parallel Patterns

- **Parallel Patterns**: A recurring combination of task distribution and data access that solves a specific problem in parallel algorithm design.
- Patterns provide us with a "vocabulary" for algorithm design
- It can be useful to compare parallel patterns with serial patterns
- Patterns are universal – they can be used in *any* parallel programming system

Some Basic Patterns

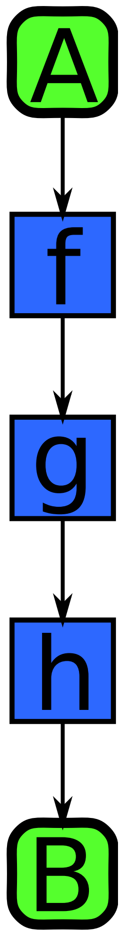
- **Serial:** Sequence

→ **Parallel:** Superscalar Sequence

- **Serial:** Iteration

→ **Parallel:** Map, Reduction, Scan, Recurrence…

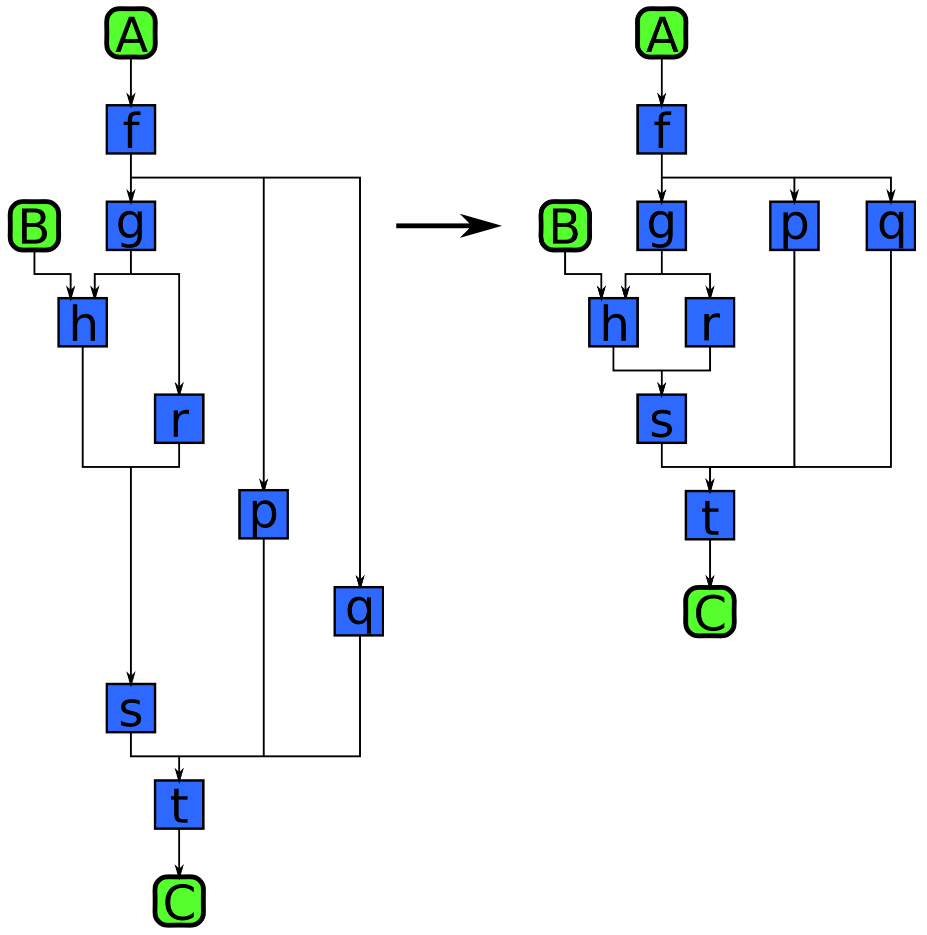
(Serial) Sequence



A serial sequence is executed in the exact order given:

```
F = f(A);
G = g(F);
B = h(G);
```

Superscalar Sequence



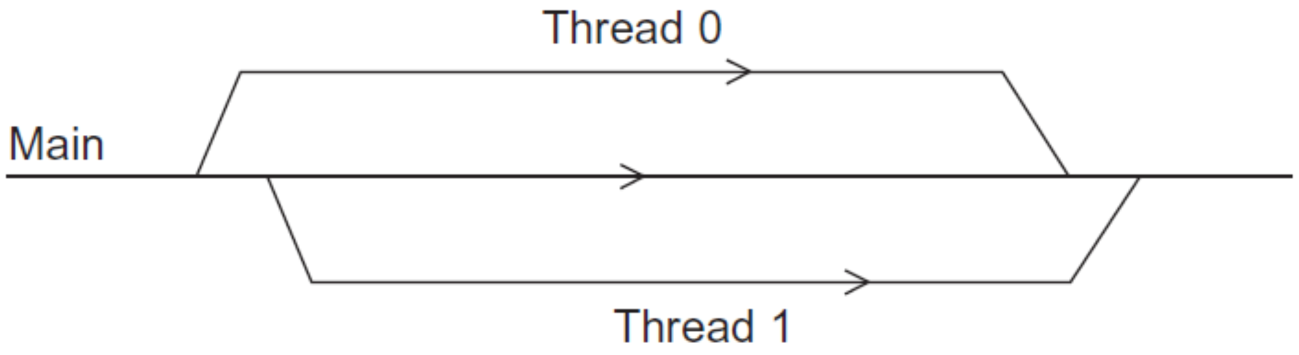
Developer writes “serial” code:

```
F = f(A);
G = g(F);
H = h(B,G);
R = r(G);
P = p(F);
Q = q(F);
S = s(H,R);
C = t(S,P,Q);
```

- Tasks ordered only by data dependencies
- Tasks can run whenever input data is ready

(Parallel) Fork-Join

- **Fork-join**: allows control flow to fork into multiple parallel flows, then rejoin later
- Pthread examples were mostly fork-join
  - Create threads to do work
  - Join them back together at the end
  - Main thread "processes" work



Fork-Join vs. Barrier

- Barrier cause all threads to “meet” then continue working independently

```
thread_func( . . . ) {
  for(int i=0; i < N i++) {
    if (i % 1000) {
      barrier();
      // Coordinate.
    }
  }
}
```

- Fork-Join only one thread continues

Futures

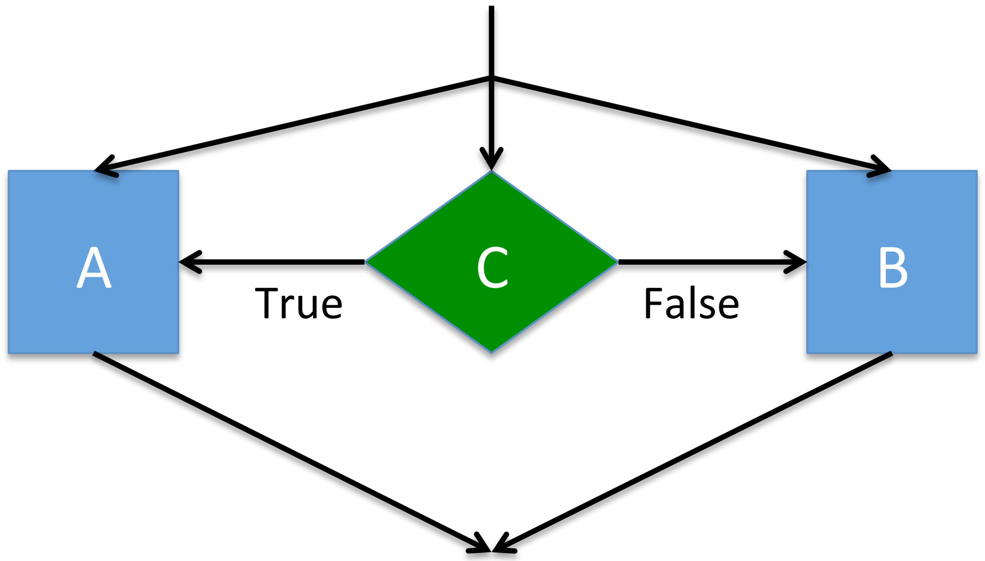
```
#include <future>
#include <iostream>

int called_from_async() {
   std::cout << "Async call" << std::endl;
   return 5;
}

int main() {
  std::future<int> fut = std::async(called_from_async);
  std::cout << "Message from main." << std::endl;
  std::cout << “Future is " << fut.get() << std::endl;
  return 0;
}
```

std::Future and std::Promise

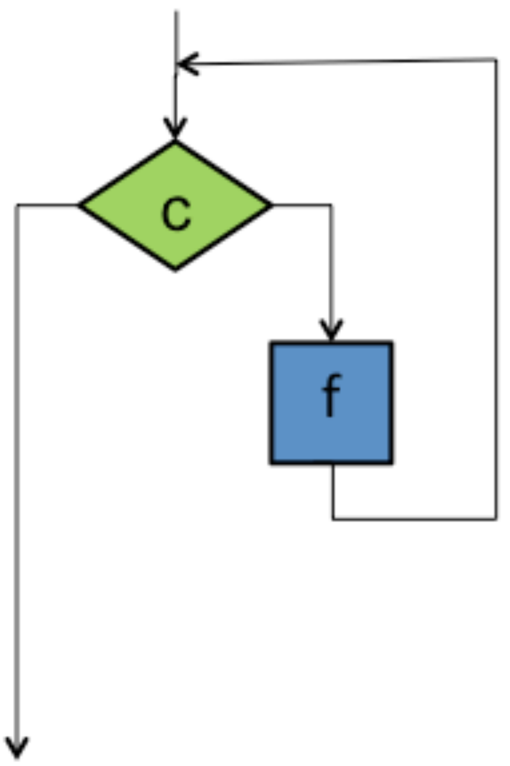
- New in c++11 standard library
- Future is a contract with program to provide a value at sometime in the future.
- Promise allows you to manage your own threads, a promise provides a future. Promises provide the “data” exchange.

Speculative Selection

- Run multiple scenarios and pick at end.
- More expensive to wait to know which branch.



(Serial) Iteration



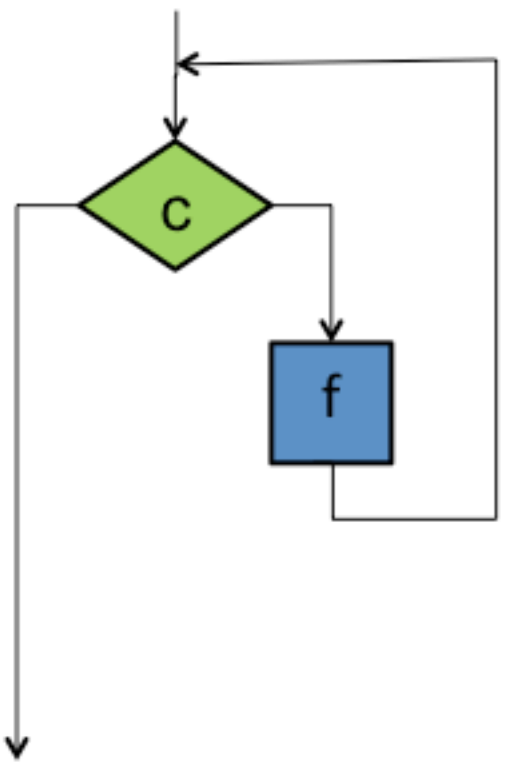
The iteration pattern repeats some section of code as long as a condition holds

```
while (c) {
   f();
}
```

Each iteration can depend on values computed in any earlier iteration.

The loop can be terminated at any point based on computations in any iteration

(Serial) Countable Iteration



The iteration pattern repeats some section of code a specific number of times

```
for (i = 0; i<n; ++i) {
    f();
}
```

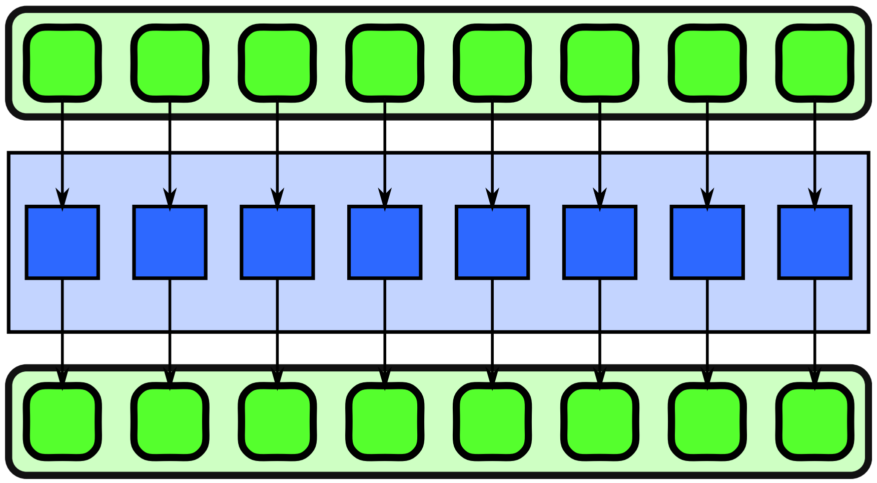
This is the same as

```
i = 0;
while (i<n) {
    f();
    ++i;
}
```

Parallel “Iteration”

- The serial iteration pattern actually maps to several *different* parallel patterns
- It depends on whether and how iterations depend on each other…
- Most parallel patterns arising from iteration require a fixed number of invocations of the body, known in advance

Map



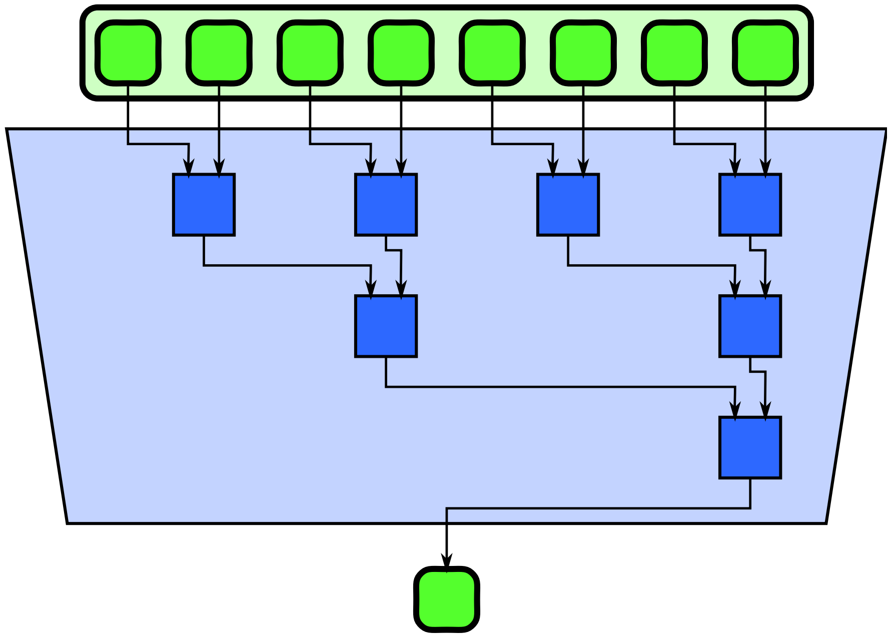
Examples: gamma correction and thresholding in images; color space conversions; Monte Carlo sampling; ray tracing.

- *Map* replicates a function over every element of an index set
- The index set may be abstract or associated with the elements of an array.

```
for (i=0; i<n; ++i) {
    f(A[i]);
}
```

- Map replaces *one specific* usage of iteration in serial programs: *independent operations.*

Reduction



- *Reduction* combines every element in a collection into one element using an associative operator.

```
b = 0;
for (i=0; i<n; ++i) {
    b += f(B[i]);
}
```

- Reordering of the operations is often needed to allow for parallelism.
- A tree reordering requires associativity.

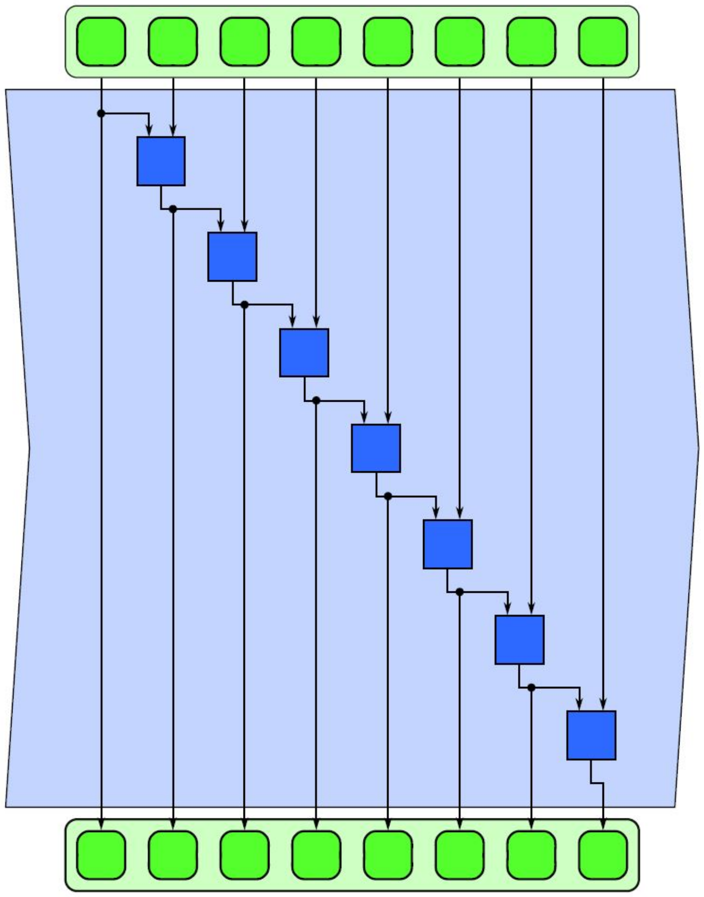
Examples: averaging of Monte Carlo samples; convergence testing; image comparison metrics; matrix operations.

Re-ordering Computions

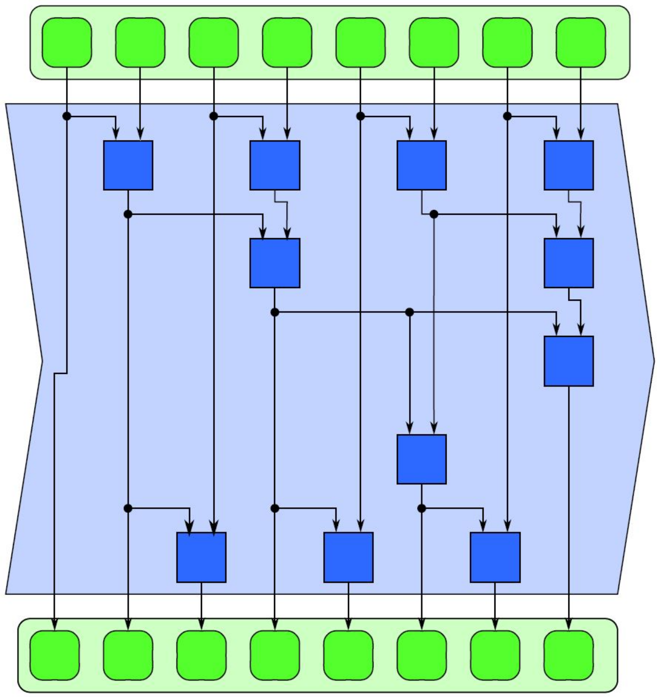
- Associative
  $(A \times B) \times C = A \times (B \times C)$
  - You can re-group but not re-order
  - Tree based reduction requires this
- Commutative
  $A \times B = B \times A$
  - You can re-group or re-order

Scan

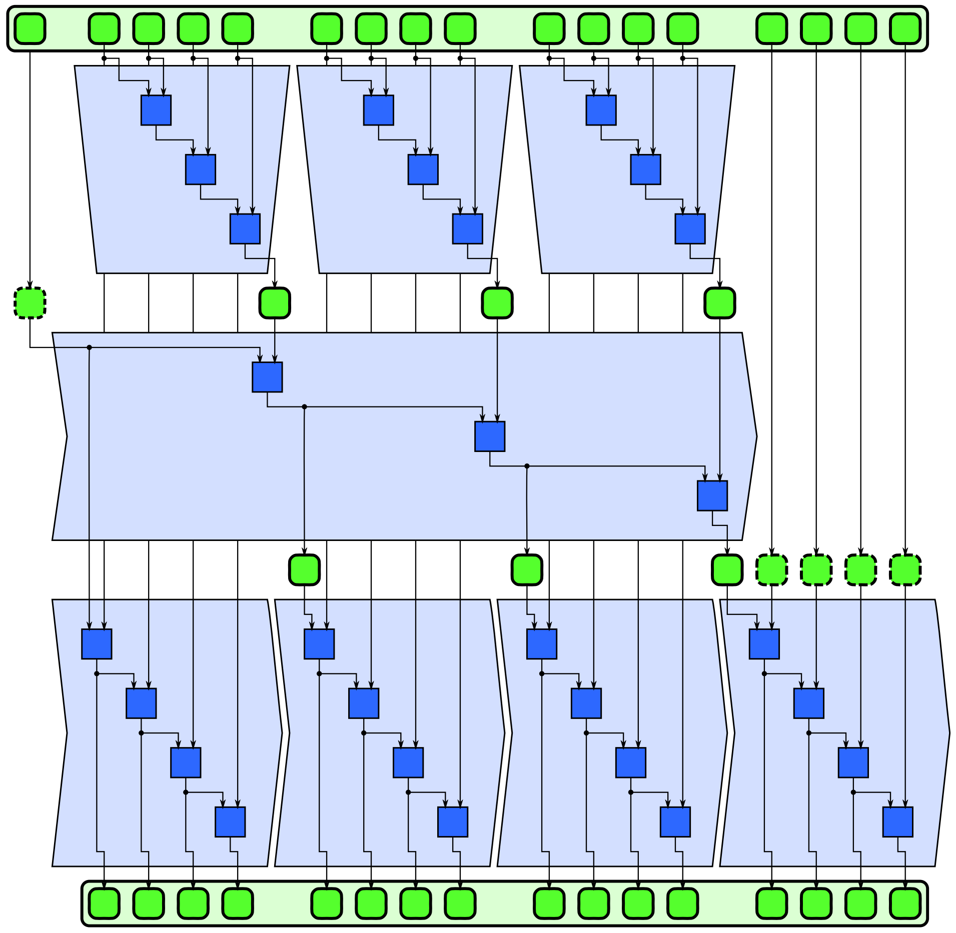
Serial Scan



Parallel Scan



Scan



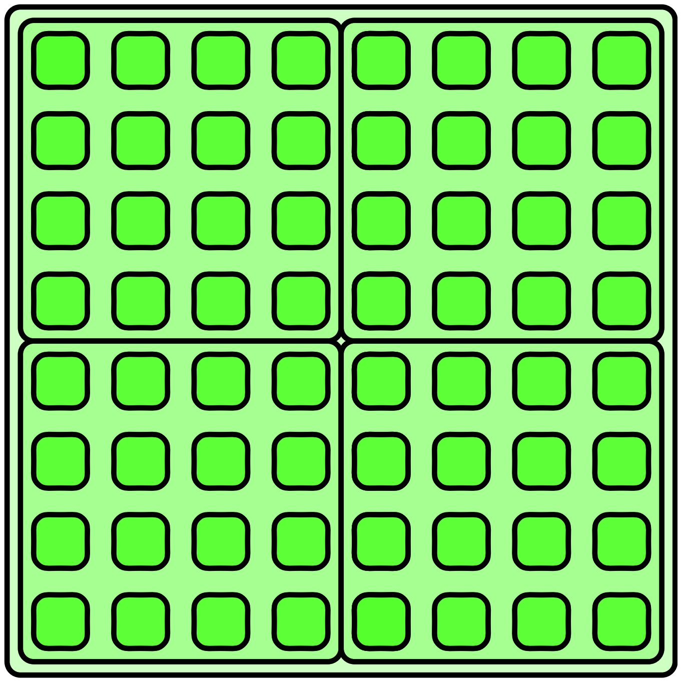
- *Scan* computes all partial reductions of a collection

```
A[0] = B[0] + init;
for (i=1; i<n; ++i) {
 A[i] = B[i] + A[i-1];
}
```

- Operator must be (at least) associative.
- Diagram shows one possible parallel implementation using three-phase strategy
- Scan is a special case of serial *fold* pattern

Examples: random number generation, pack, tabulated integration, time series analysis

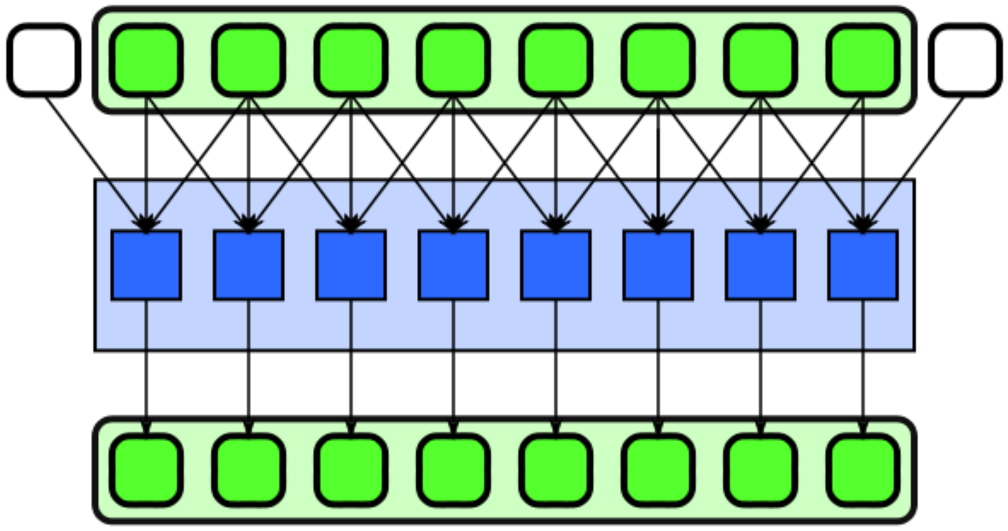
Geometric Decomposition/Partition



- *Geometric decomposition* breaks an input collection into sub-collections
- *Partition* is a special case where sub-collections do not overlap
- Does not move data, it just provides an alternative "view" of its organization

Examples: JPG and other macroblock compression; divide-and-conquer matrix multiplication; coherency optimization for cone-beam recon.

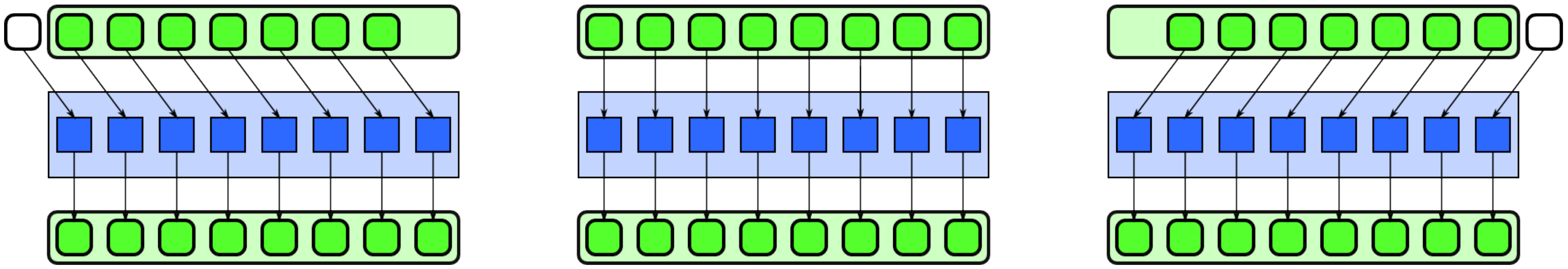
Stencil

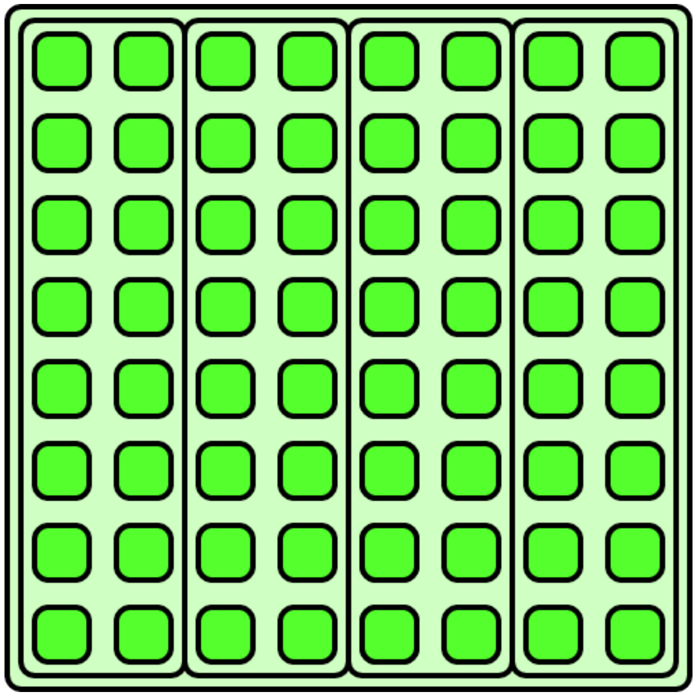


Examples: signal filtering including convolution, median, anisotropic diffusion

- *Stencil* applies a function to neighbourhoods of a collection.
- Neighbourhoods are given by set of relative offsets.
- Boundary conditions need to be considered, but majority of computation is in interior.

Implementing Stencil

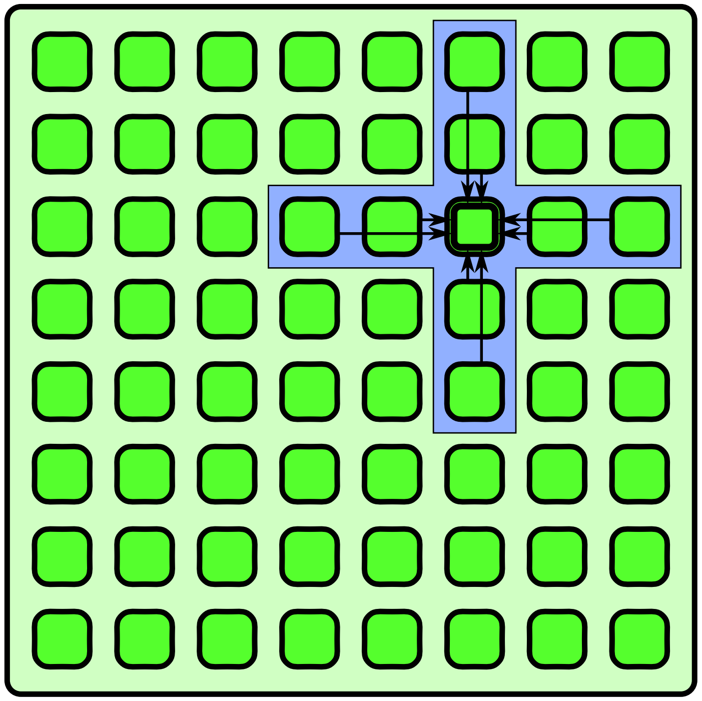




Vectorization can include converting regular reads into a set of shifts.

Strip-mining reuses previously read inputs within serialized chunks.

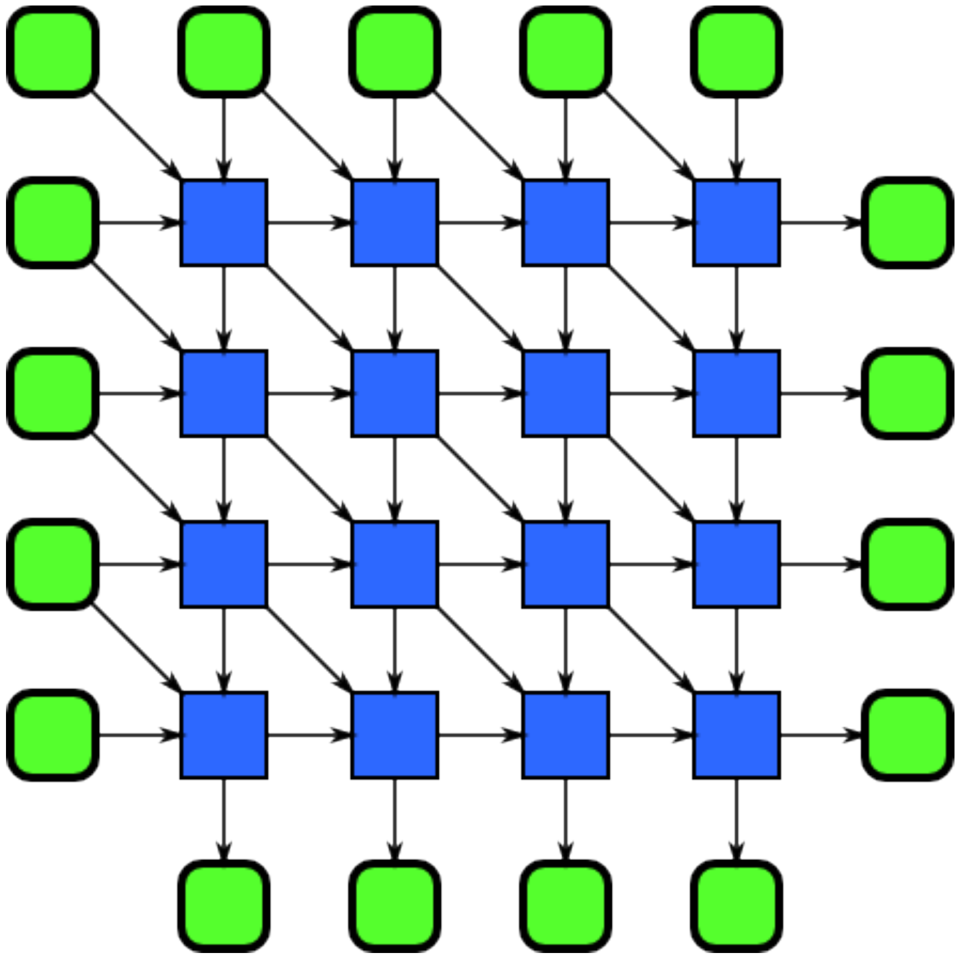
nD Stencil



- *nD Stencil* applies a function to neighbourhoods of an nD array
- Neighbourhoods are given by set of relative offsets
- Boundary conditions need to be considered

Examples: image filtering including convolution, median, anisotropic diffusion; simulation including fluid flow, electromagnetic, and financial PDE solvers, lattice QCD

Recurrence



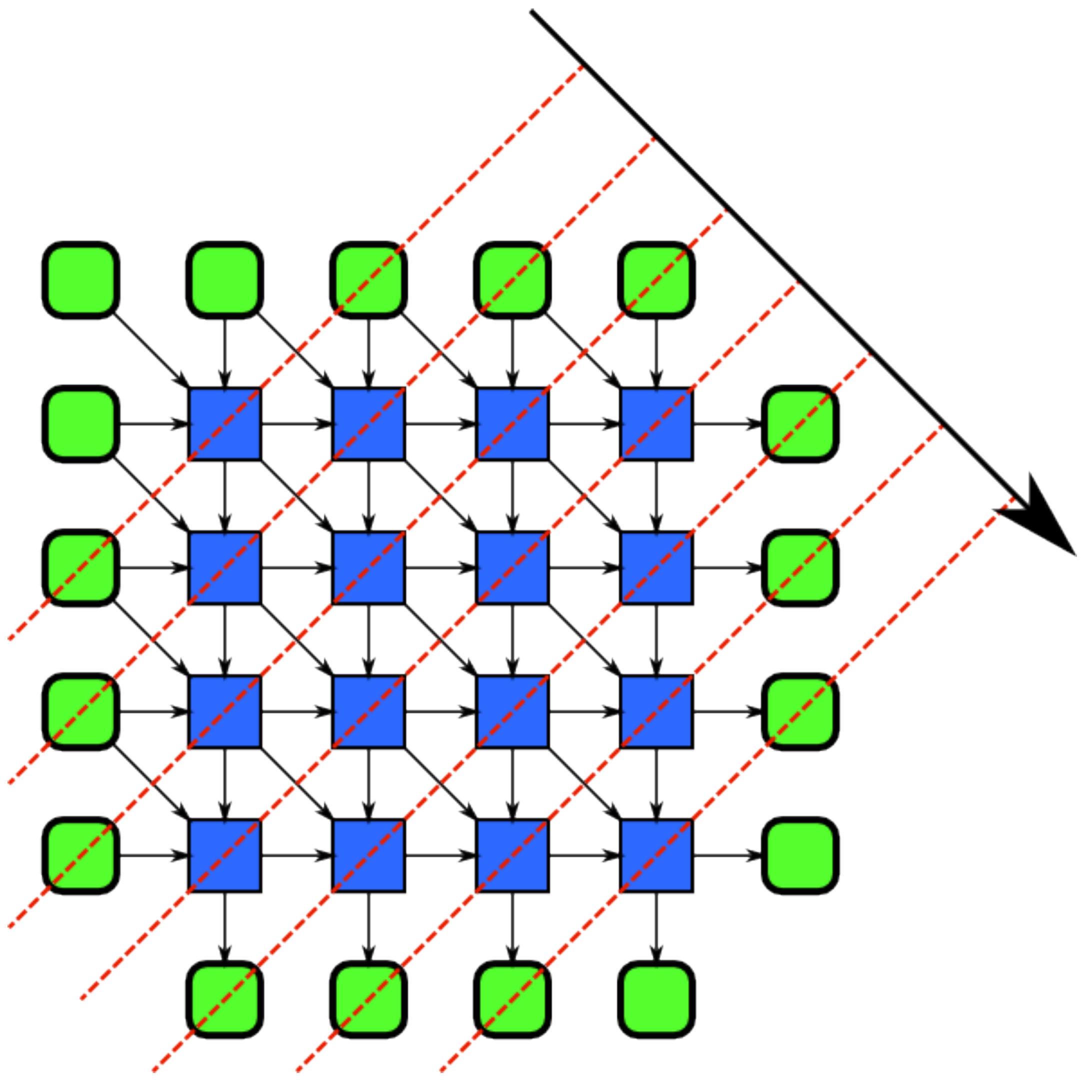
- *Recurrence* results from loop nests with both input and output dependencies between iterations
- Can also result from iterated stencils

Examples: Simulation including fluid flow, electromagnetic, and financial PDE solvers, lattice QCD, sequence alignment and pattern matching

Recurrence

```
for (int i = 1; i < N; i++) {
  for (int j = 1; j < M; j++) {
    A[i][j] = f(
      A[i-1][j],
      A[i][j-1],
      A[i-1][j-1]);
   }
}
```

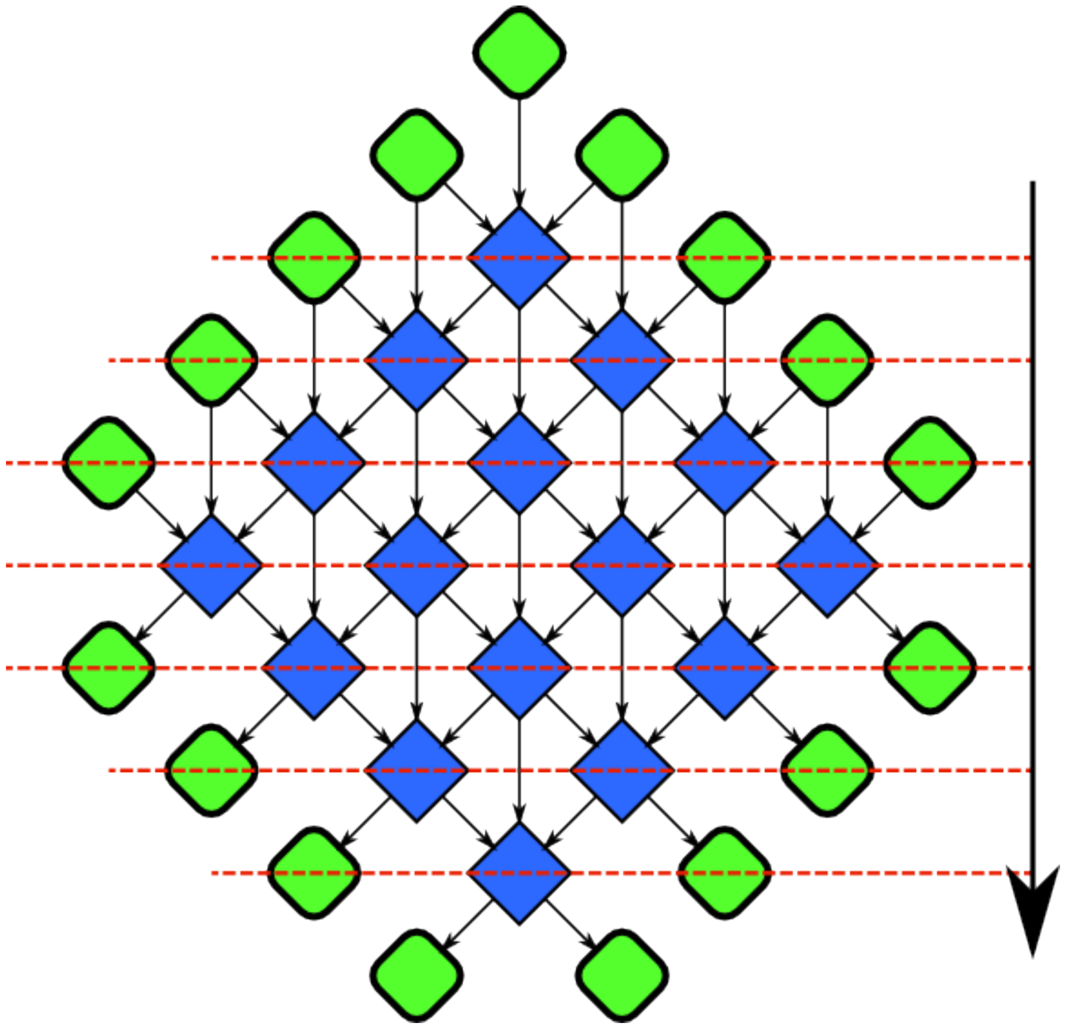
Recurrence Hyperplane Sweep



- Multidimensional recurrences can *always* be parallelized

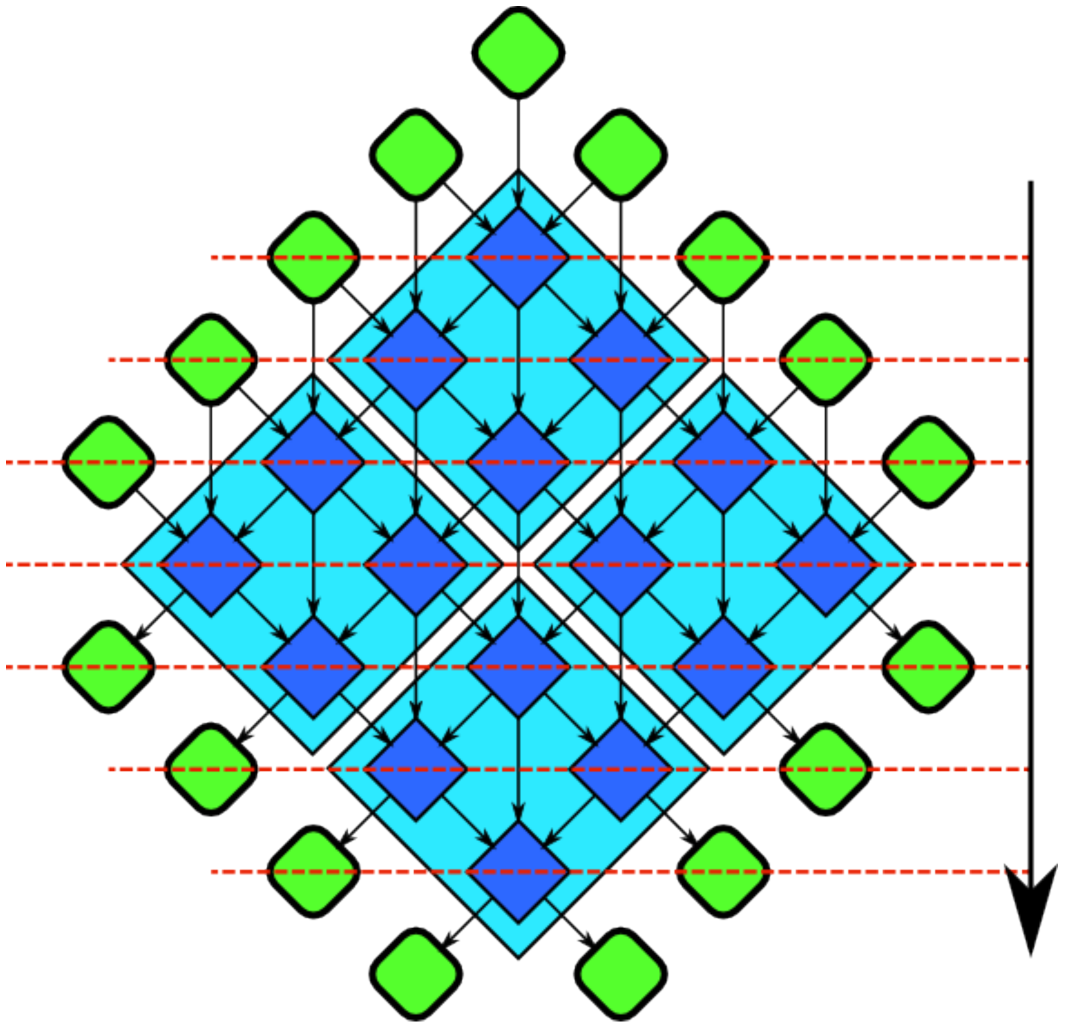
- Leslie Lamport's hyperplane separation theorem:
  - Choose hyperplane with inputs and outputs on opposite sides
  - Sweep through data perpendicular to hyperplane

Rotated Recurrence



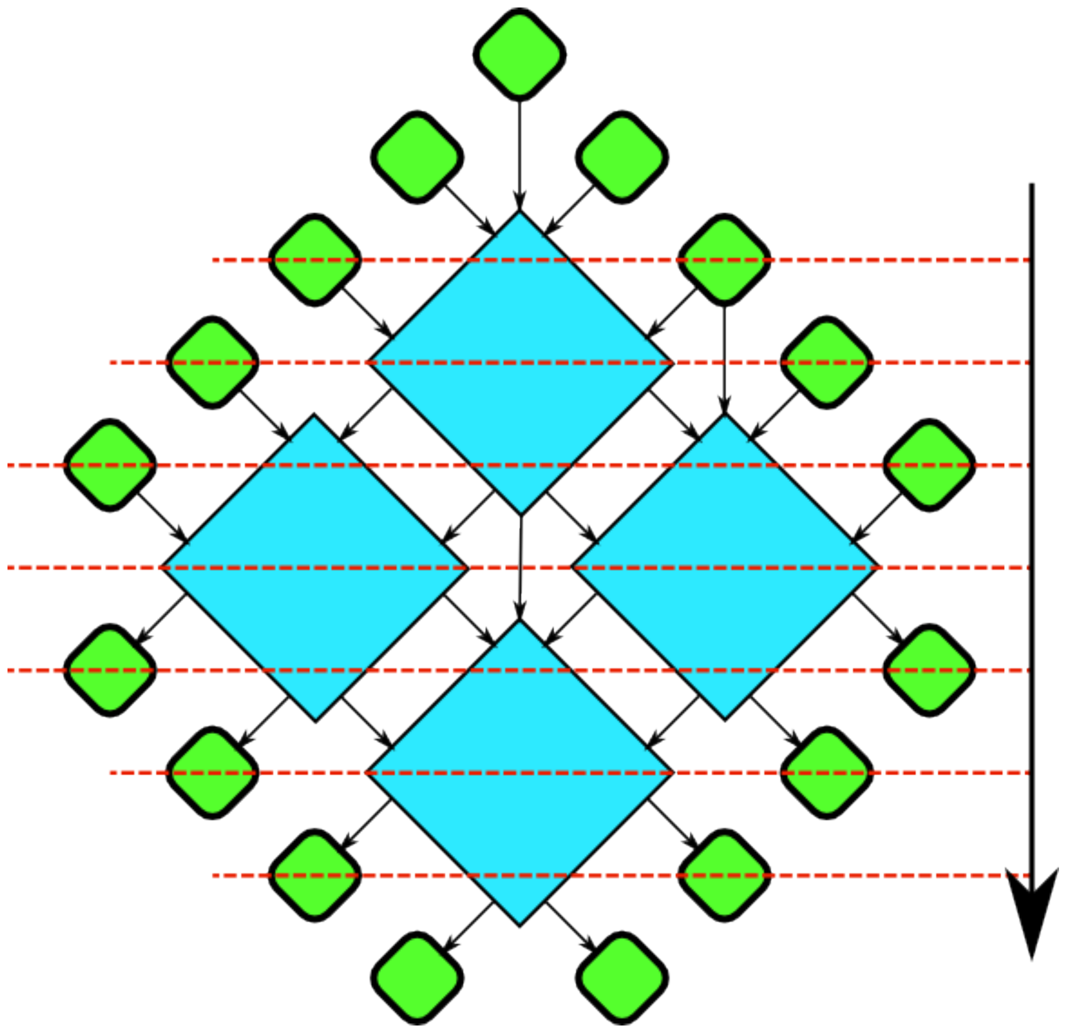
- Rotate recurrence to see sweep more clearly

Tiled Recurrence



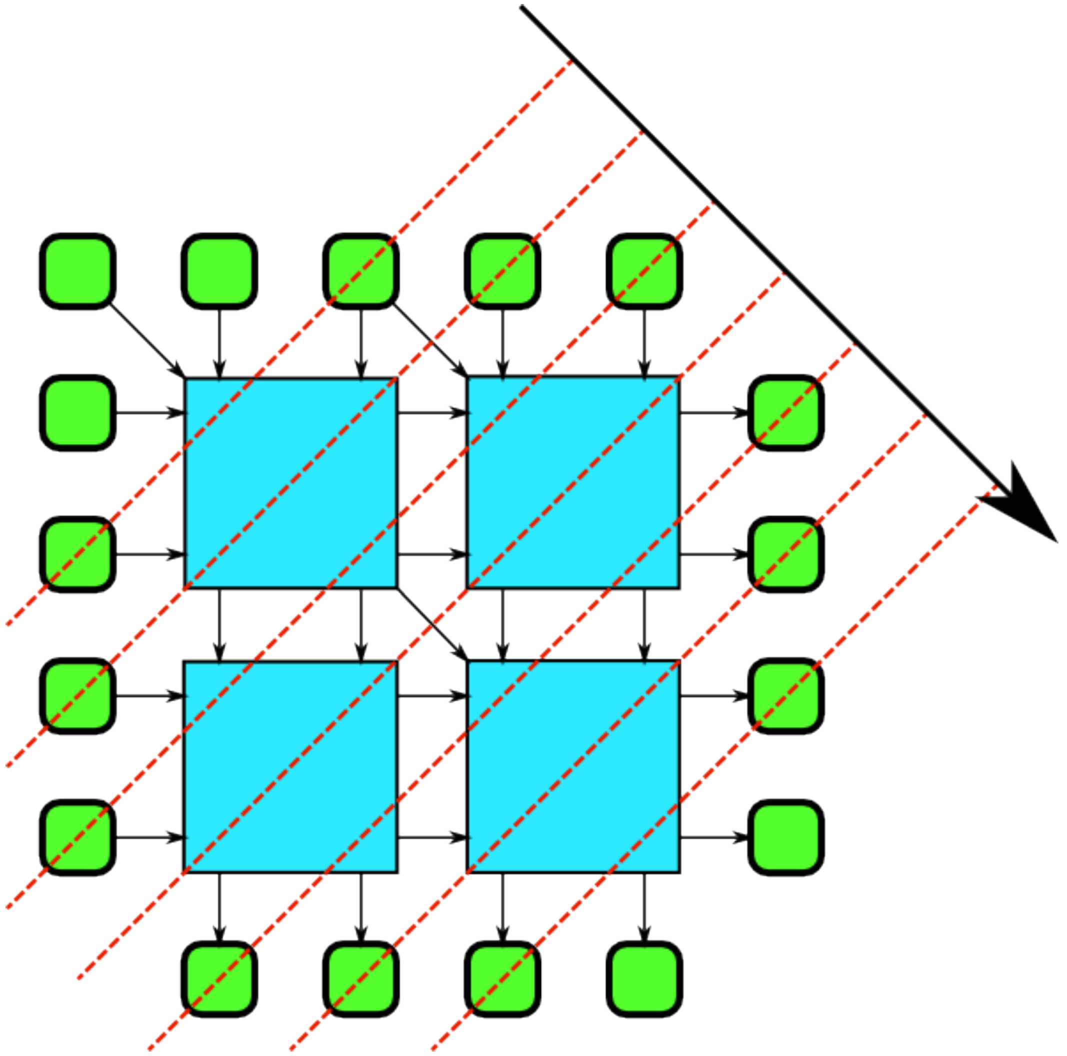
- Can partition recurrence to get a better compute vs. bandwidth ratio
- Show diamonds here, could also use paired trapezoids

Tiled Recurrence



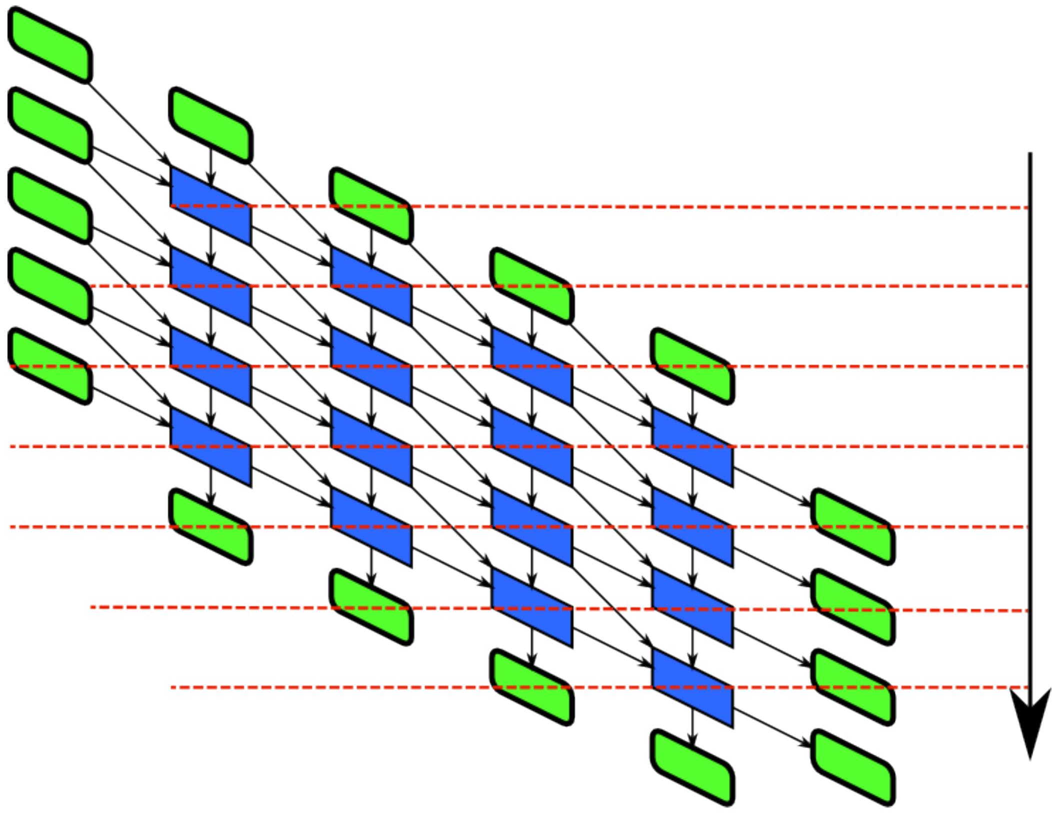
- Remove all non-redundant data dependences

Recursively Tiled Recurrences



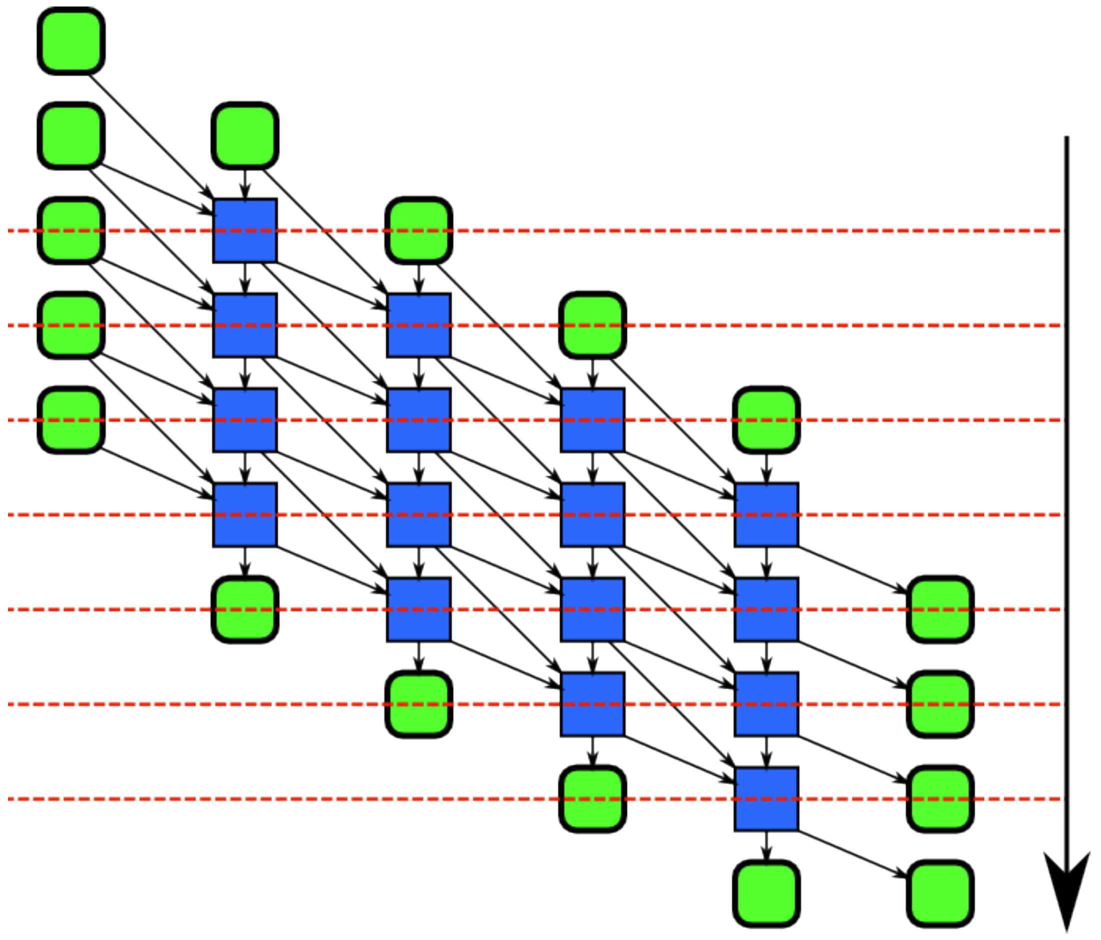
- Rotate back: same recurrence at a different scale!
- Leads to recursive cache-oblivious divide-and-conquer algorithm
- Implement with fork-join.

Skewed Recurrence



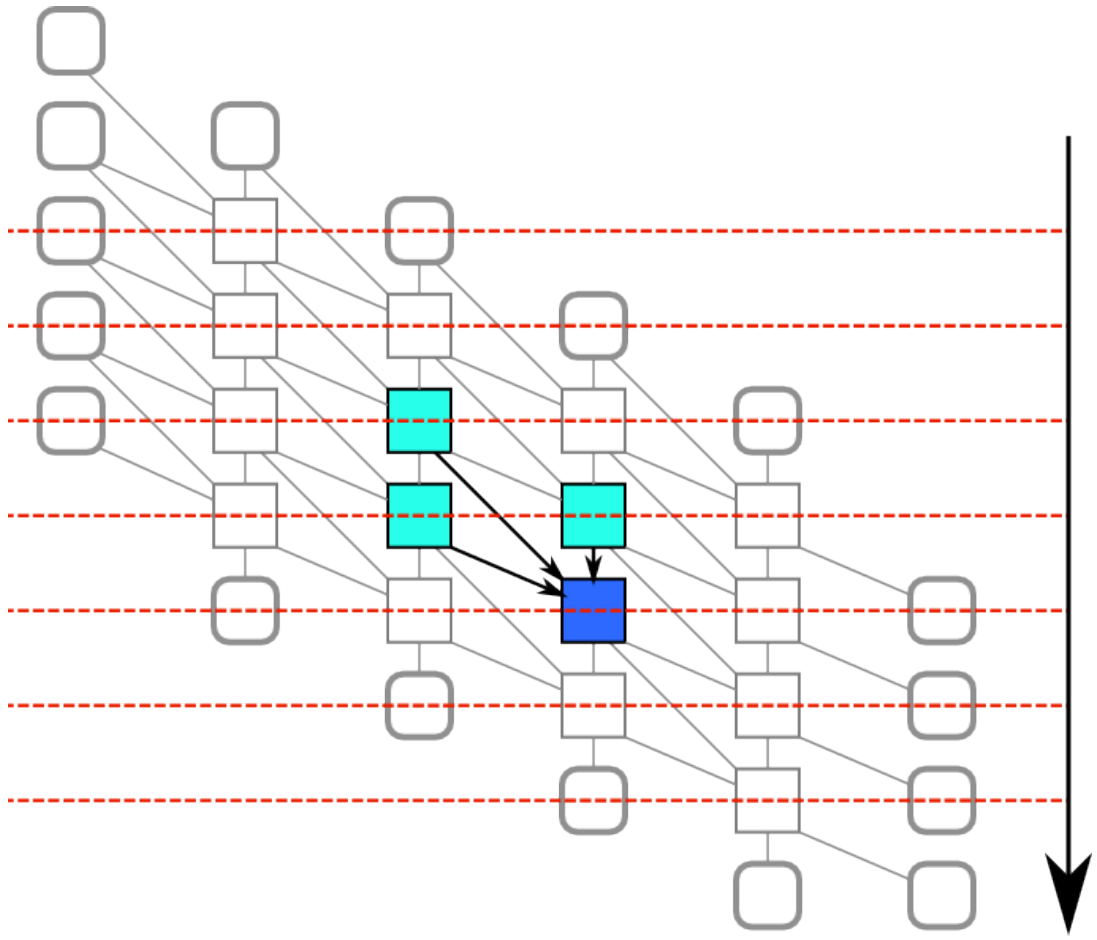
- Let's just clean up the diagram a little bit…
    - Straighten up the symbols
    - Leave the data dependences as they are

Skewed Recurrence



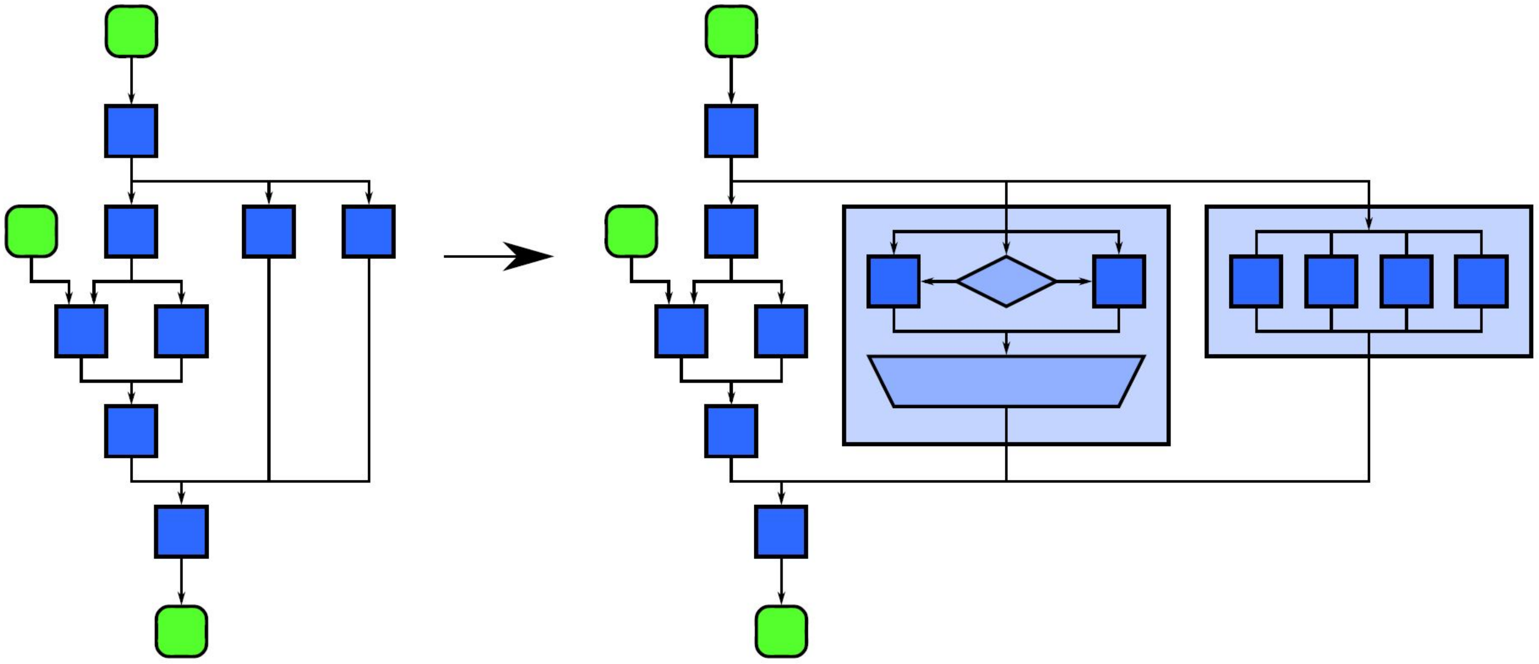
- This is a useful memory layout for implementing recurrences
- Let's now focus on one element
- Look at an element away from the boundaries

Recurrence = Iterated Stencil



- Each element depends on certain others in previous iterations
- An iterated stencil!
- *Convert iterated stencils into tiled recurrences for efficient implementation*

Nesting Pattern



Nesting Pattern: A compositional pattern. Nesting allows other patterns to be composed in a hierarchy so that any task block in the above diagram can be replaced with a pattern with the same input/output and dependencies.

(Serial) Recursion

- **Recursion**: dynamic form of nesting allowing functions to call themselves
  - Tail recursion is a special recursion that can be converted into iteration
  - Replace recursive stack with new call.

Tail Recursion Example

- Factorial
  1! = 1
  2! = 2 * 1!
  3! = 3 * 2!
  n! = n * (n-1)!

```
unsigned int fact(unsigned int n)
{
    if (n == 0) return 1;

    return n*fact(n-1);
}
```

Tail Recursion Example

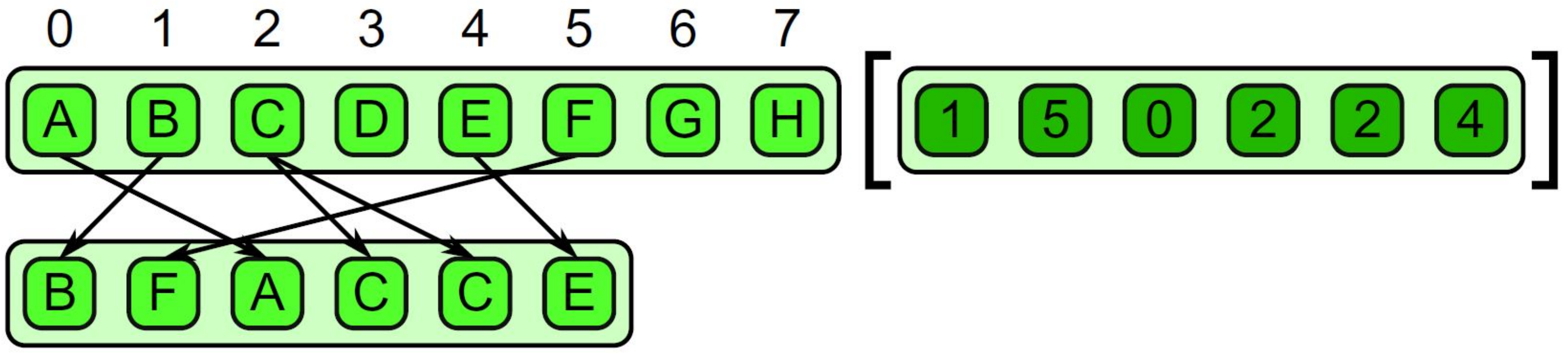
```
unsigned int factTail(
       unsigned int n, unsigned int a)
{
    if (n == 0)  return a;
    return factTR(n-1, n*a);
}
unsigned int fact(unsigned int n)
{
   return factTail(n, 1);
}
```

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**

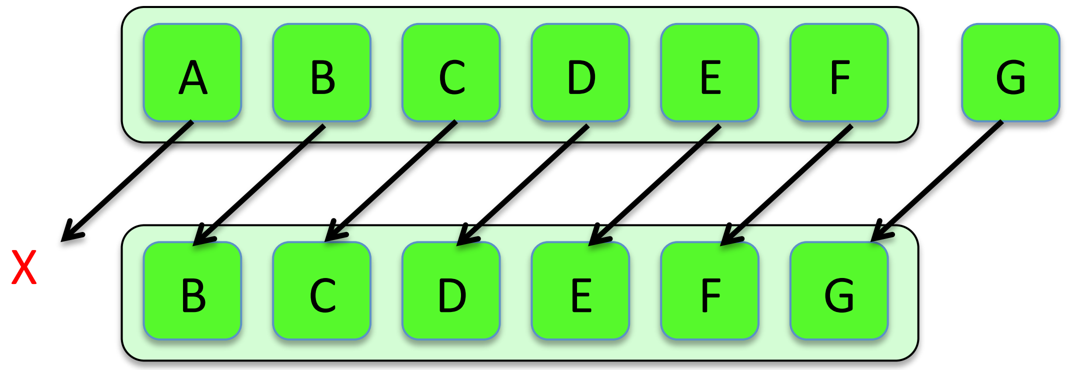
Parallel Data Management Patterns: Gather

- **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



Shift

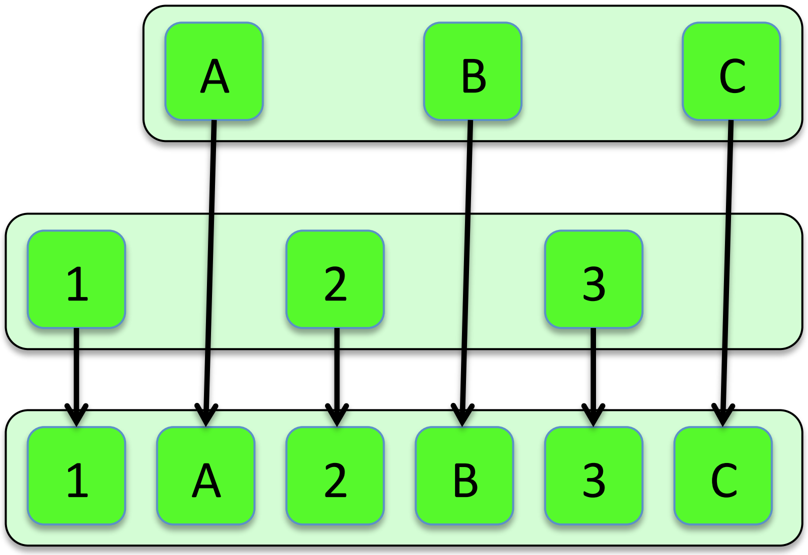
- Just a special version of a gather



- Variants depending on how you handle boundary conditions.

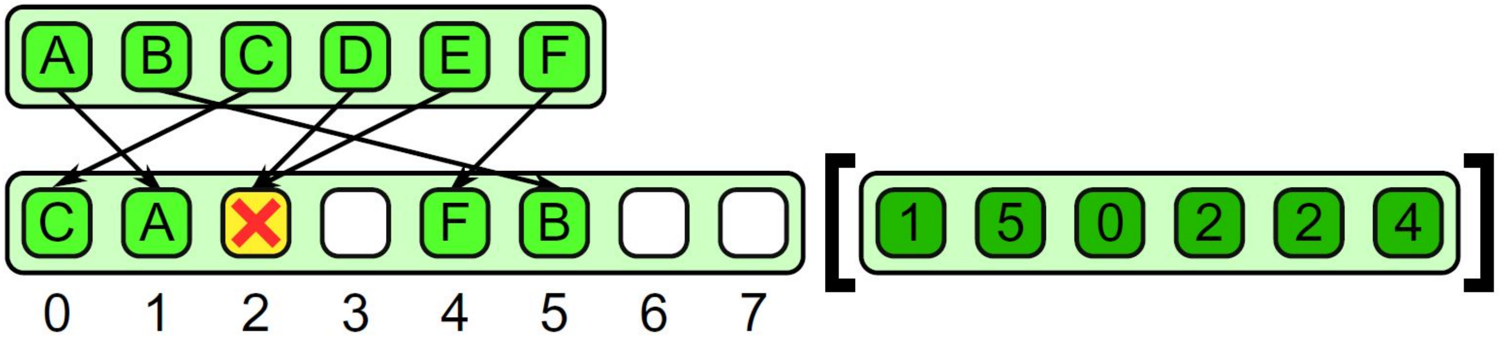
Zip and Unzip

- Another special version of gather



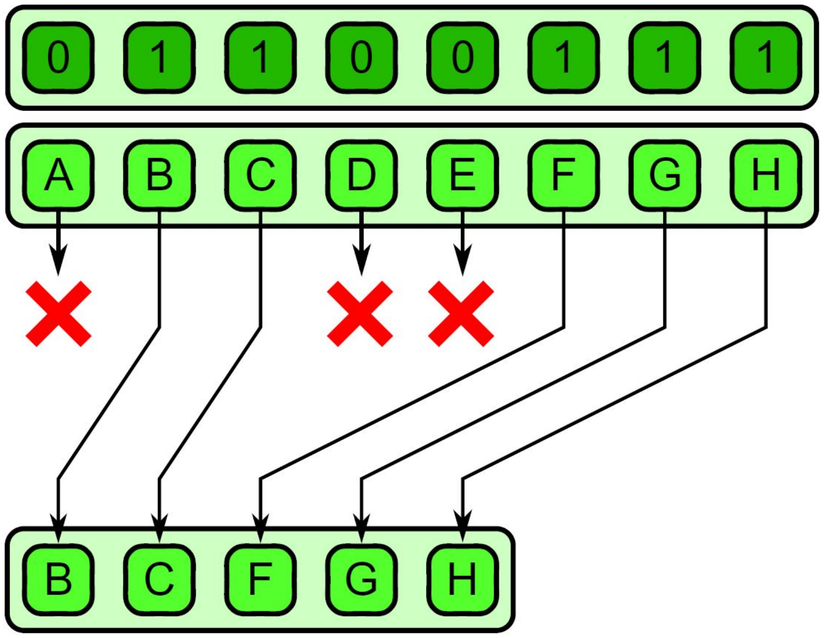
Parallel Data Management Patterns: Scatter

- **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!



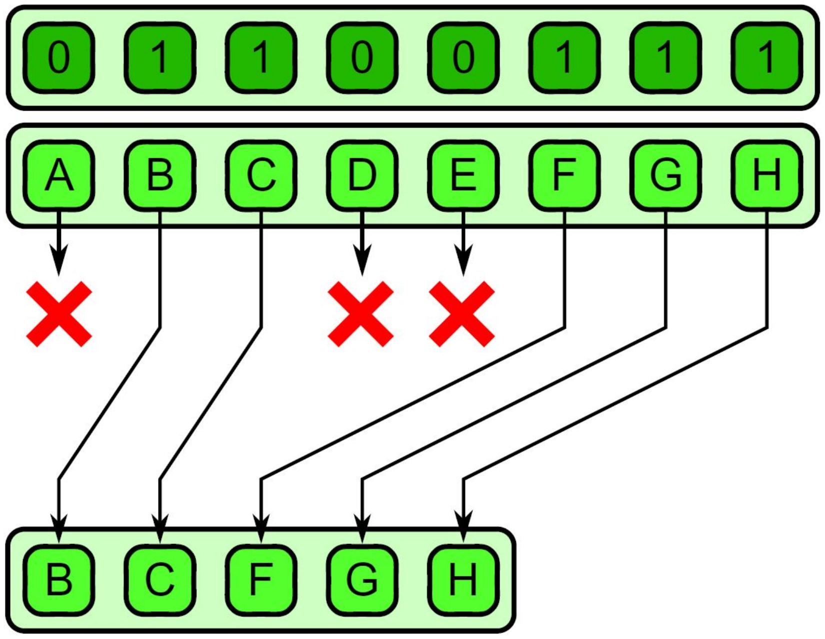
Parallel Data Management Patterns: Pack

- **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



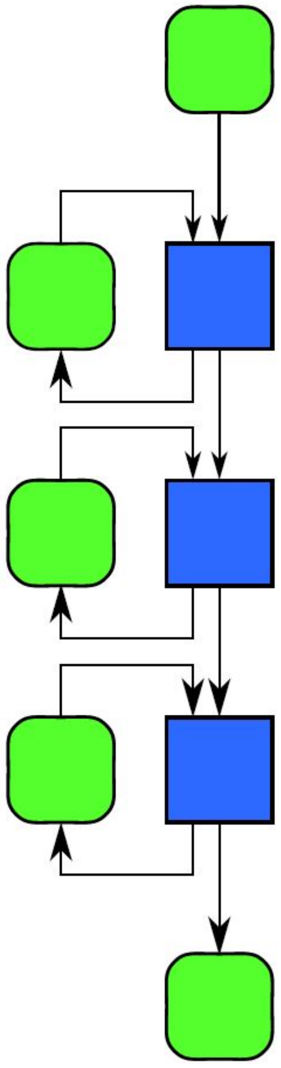
Implementation of Pack

- How would you implement pack?

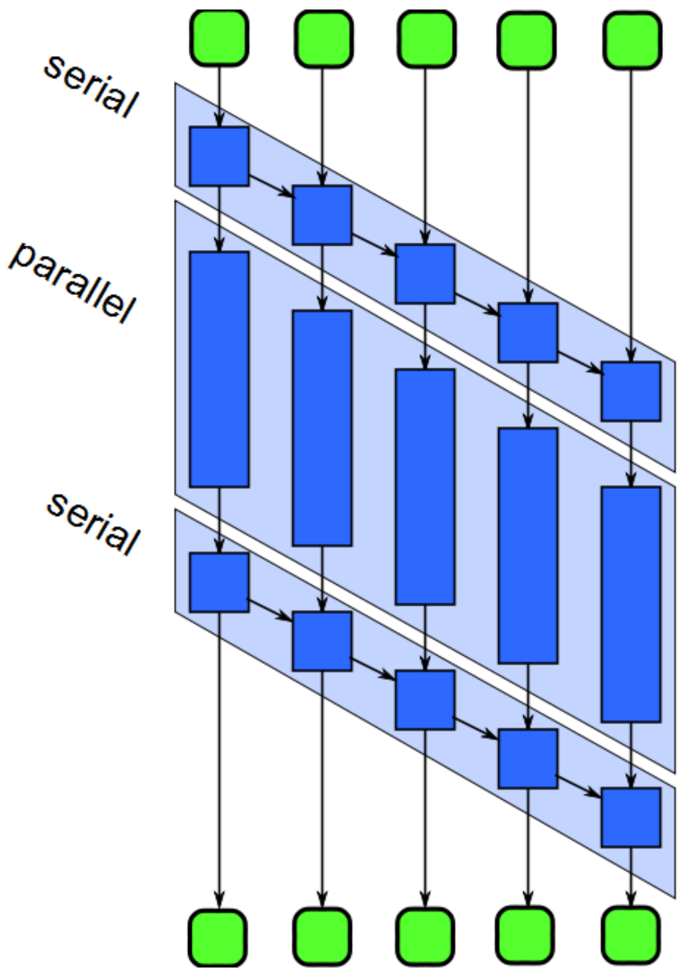


Parallel Data Management Patterns: Pipeline

- **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



Pipeline



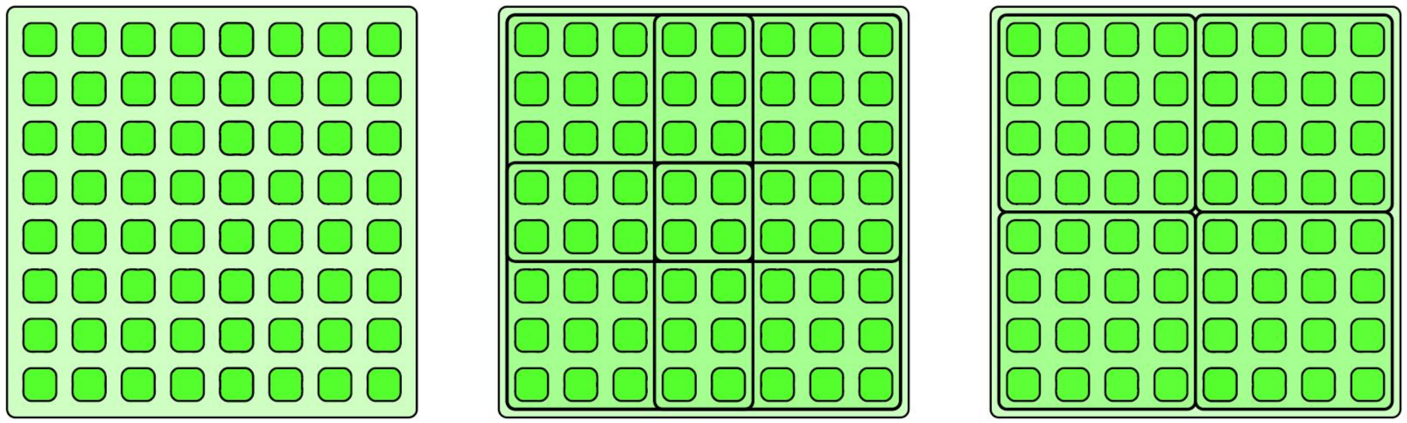
- Parallelize pipeline by
  - Running different stages in parallel
  - Running *multiple copies* of stateless stages in parallel

- Running multiple copies of stateless stages in parallel requires reordering of outputs

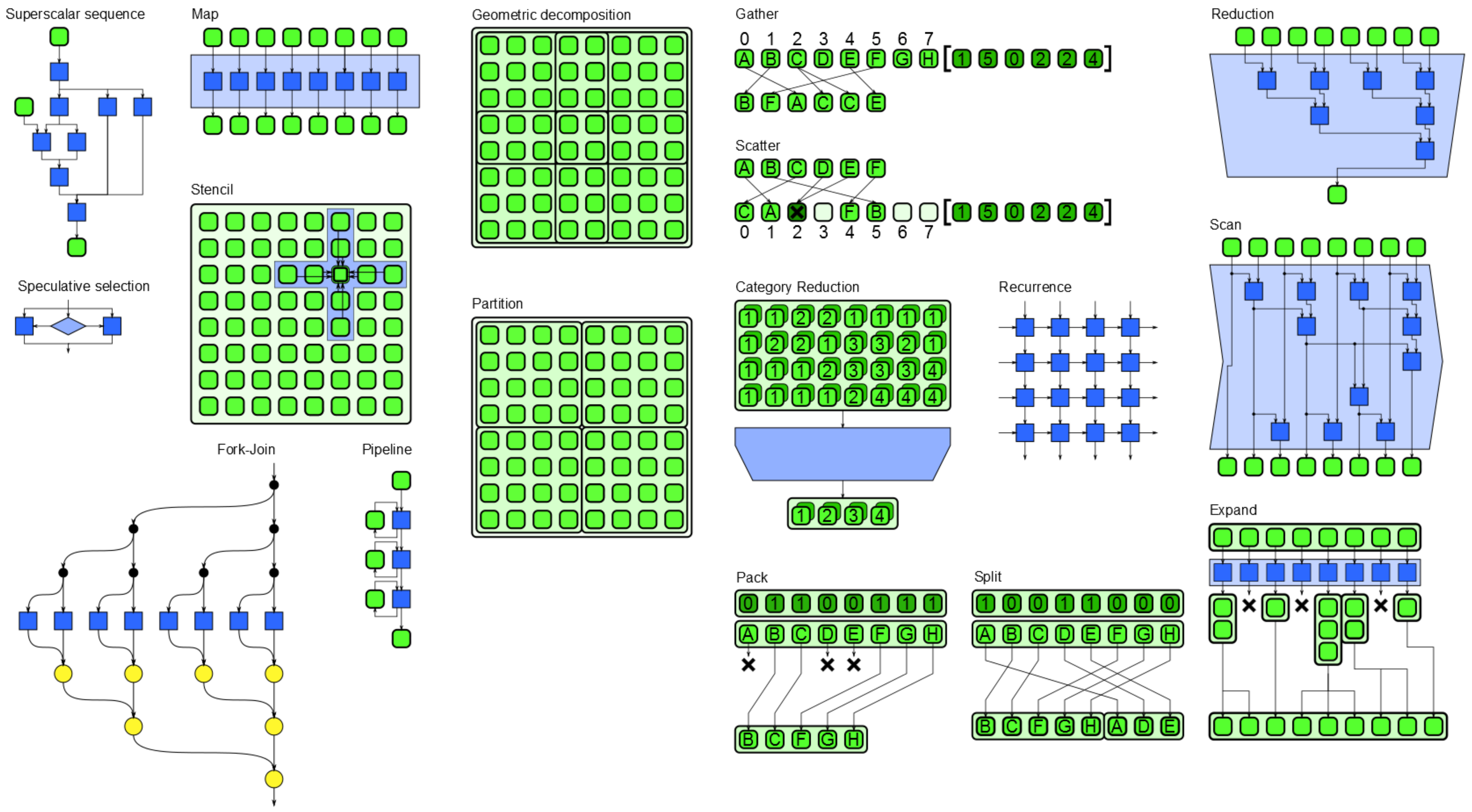
- Need to manage buffering between stages

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 Patterns: Overview



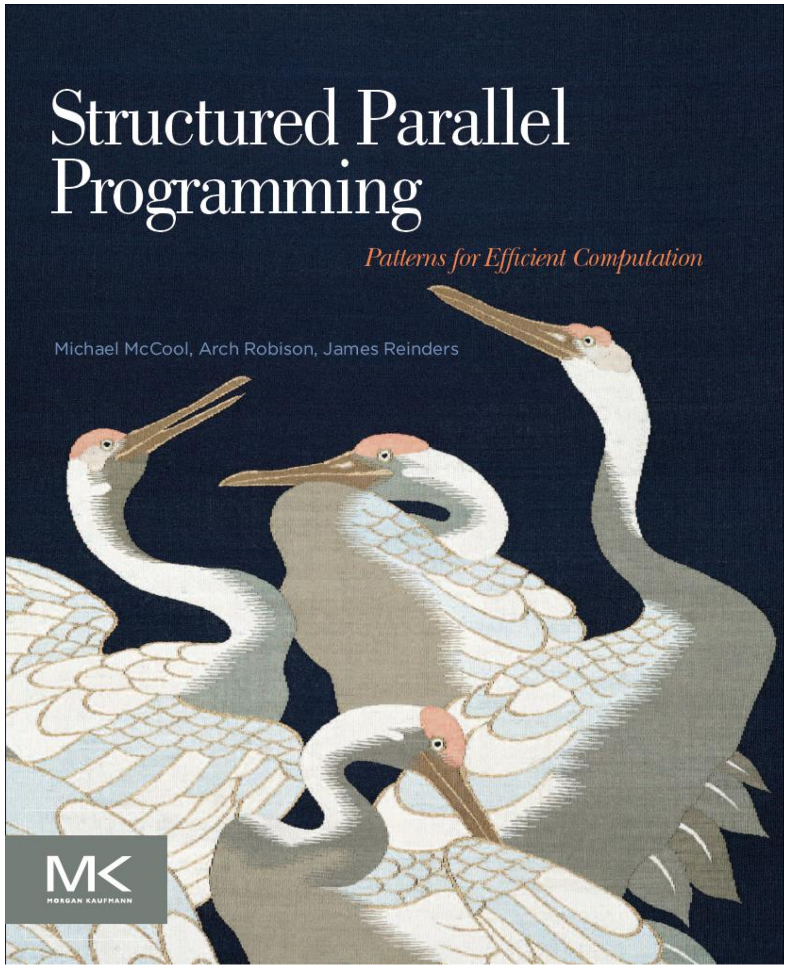
Citation

Slides for this lecture are based upon the following:

Structured Parallel Programming: Patterns for Efficient Computation

- Michael McCool
- Arch Robison
- James Reinders

- www.parallelbook.com


Structured Parallel Programming
Patterns for Efficient Computation
Michael McCool, Arch Robison, James Reinders
MK
MORGAN KAUFMANN