Structured Parallel Programming

CSE 531

Spring 2023

Mahmut Taylan Kandemir

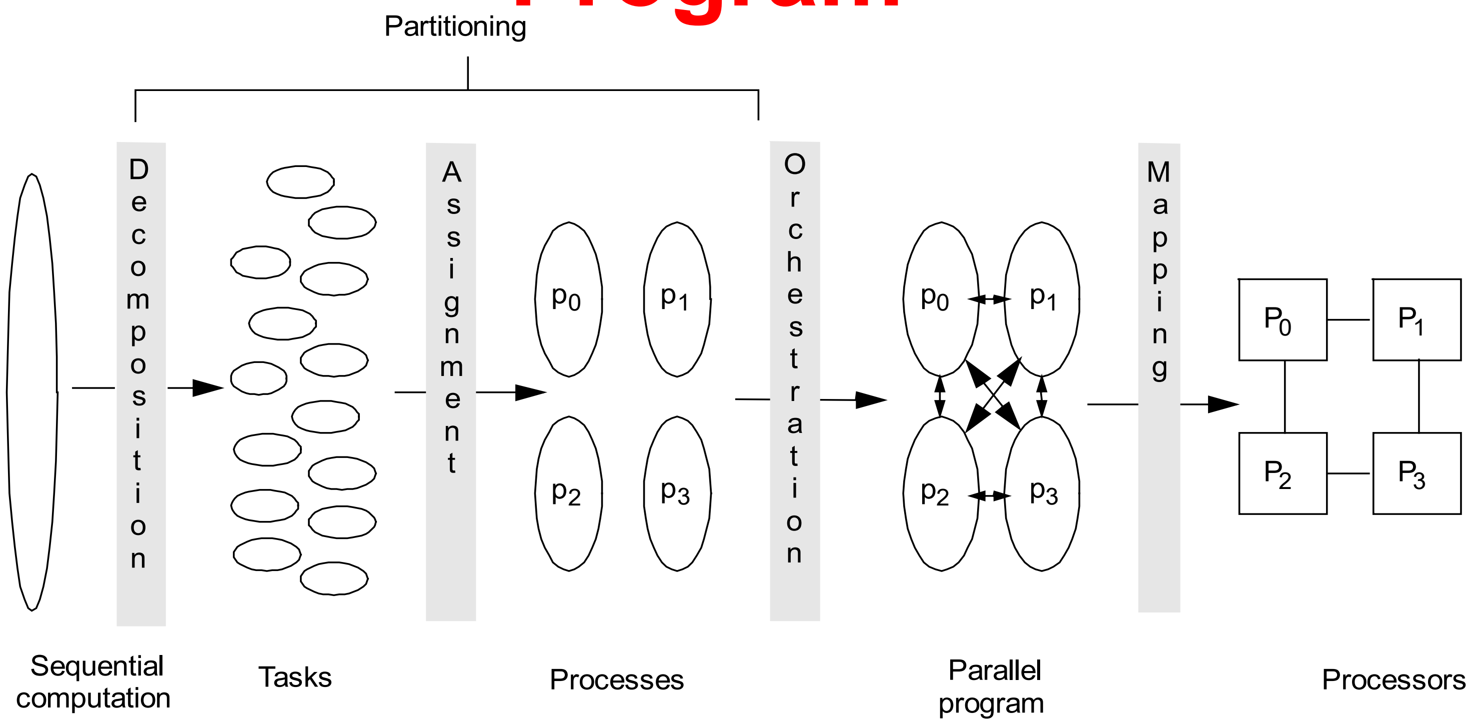
Parallelism and Performance

There are limits to "automatic" improvement of scalar performance:

1. **The Power Wall:** Clock frequency cannot be increased without excessive cooling costs.
2. **The Memory Wall:** Access to data is a limiting factor.
3. **The ILP Wall:** All the existing instruction-level parallelism (ILP) is already being used.

→ **Conclusion:** Explicit parallel mechanisms and explicit parallel programming are *required* for performance scaling.

4 Steps in Creating a Parallel Program



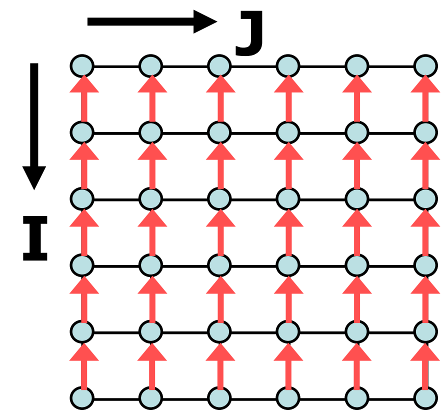
- Decomposition of computation in tasks
- Assignment of tasks to processes
- Orchestration of data access, communication, synchronization
- Mapping processes to processors

From J. P. Singh

Loop-Level Parallelism

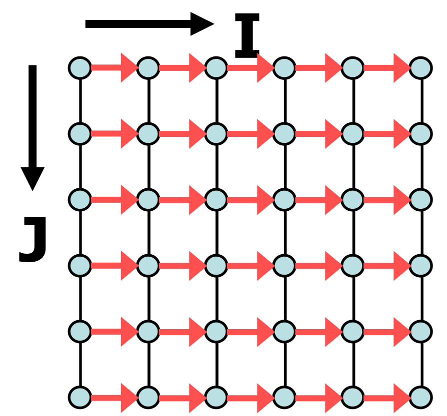
- Example

```
FOR i = 1 to N-1
   FOR j = 1 to N-1
      A[i,j] = A[i,j] + A[i-1,j];
```



- After Loop Transpose

```
FOR j = 1 to N-1
   FOR i = 1 to N-1
      A[i,j] = A[i,j] + A[i-1,j];
```



- Gets mapped into

```
Barrier();
FOR j = 1+ myPid*Iters to MIN((myPid+1)*Iters, n-1)
   FOR i = 1 to N-1
      A[i,j] = A[i,j] + A[i-1,j];
Barrier();
```

Structured Programming with Patterns

- Patterns are “best practices” for solving specific problems.
- Patterns can be used to organize your code, leading to algorithms that are more scalable and maintainable.
- A pattern supports a particular “algorithmic structure” with an efficient implementation.
- Good parallel programming models support a set of useful parallel patterns with low-overhead implementations.

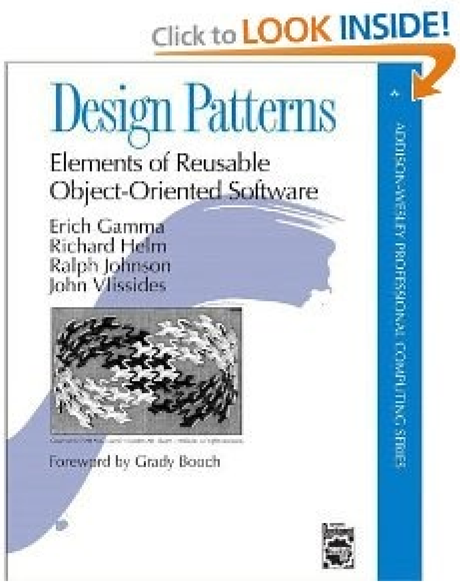
Structured Serial Patterns

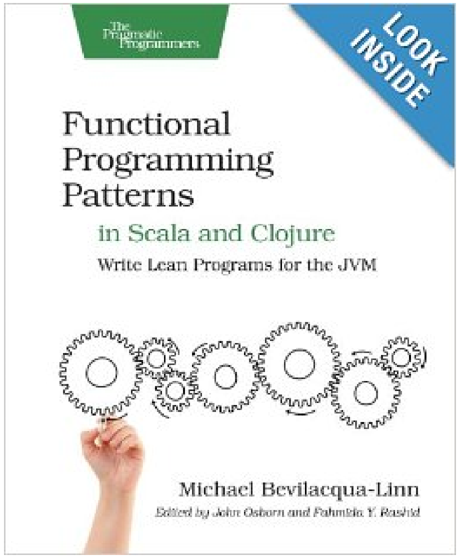
The following patterns are the basis of "**structured programming**" for serial computation:

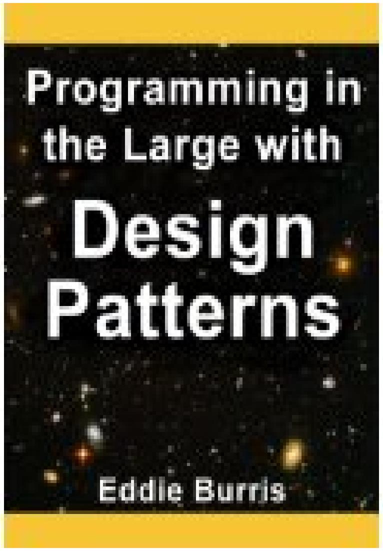
- Sequence
- Selection
- Iteration
- Nesting
- Functions
- Recursion
- Random read
- Random write
- Stack allocation
- Heap allocation
- Objects
- Closures

Using these patterns, "goto" can (mostly) be eliminated and the maintainability of software improved.

Programming with Patterns







Structured 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 intrinsics can (mostly) be eliminated and the maintainability of software improved.

Some Basic Patterns

- **Serial:** Sequence

→ **Parallel:** Superscalar Sequence

- **Serial:** Iteration

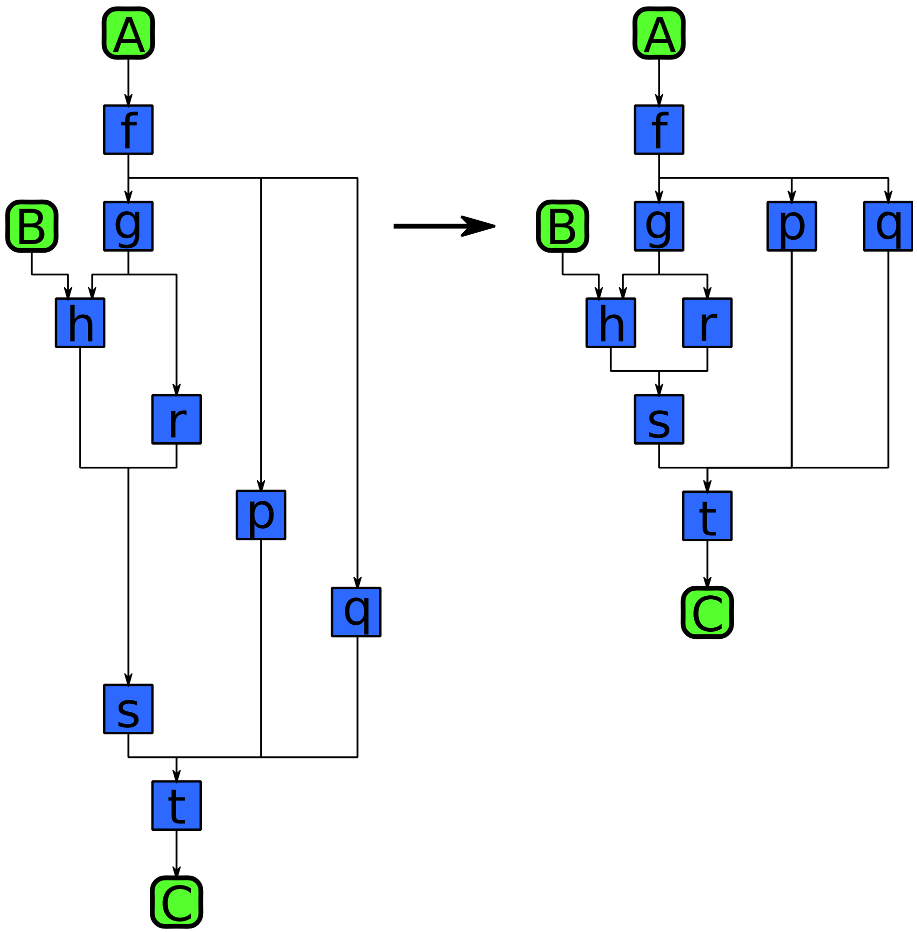
→ **Parallel:** Map, Reduction, Scan, Recurrence…

(Serial) Sequence

A serial sequence is executed in the exact order given:

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

Superscalar Sequence

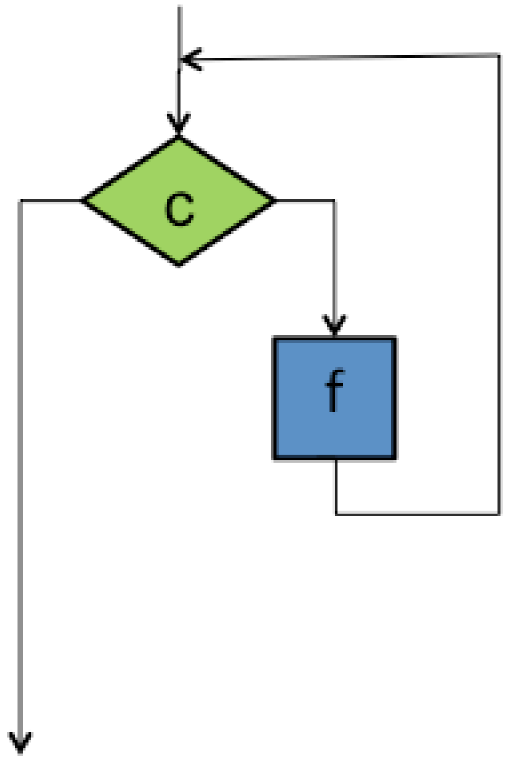


Developer writes "serial" code:

```
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

(Serial) Iteration



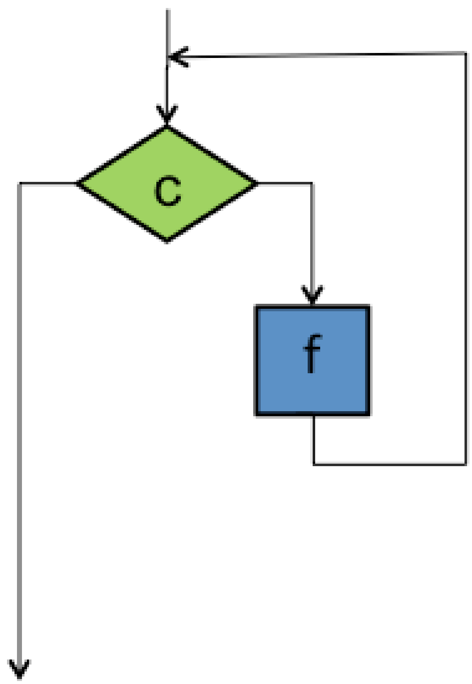
The iteration pattern repeats some section of code as long as a condition holds

```
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();
}
```

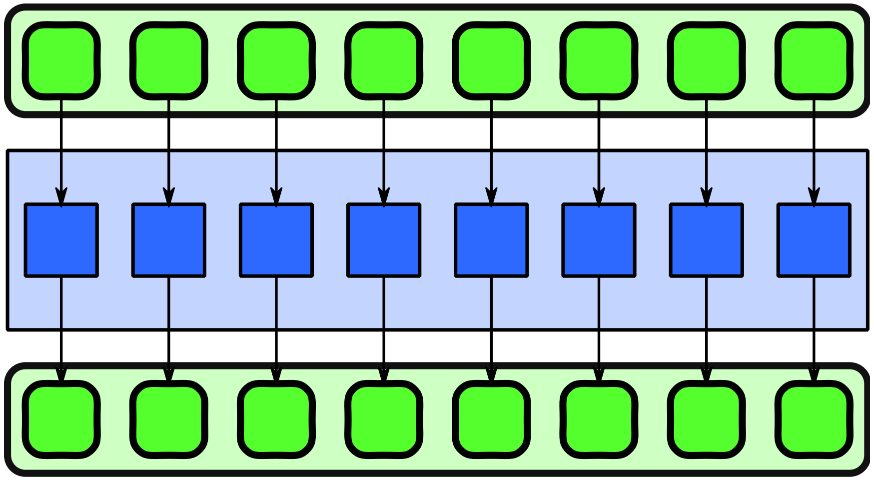
This is the same as

```
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



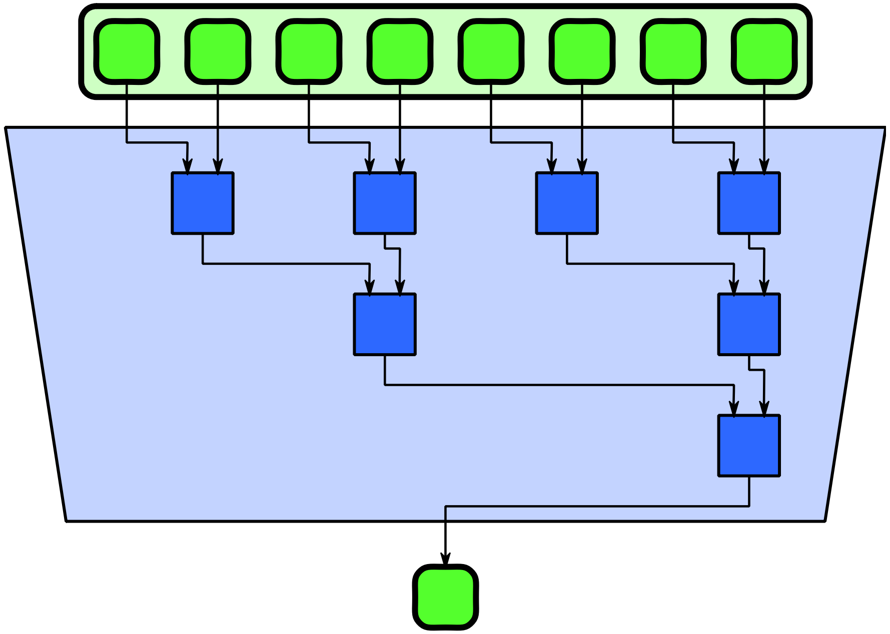
- *Map* replicates a function over every element of an index set
- The index set may be abstract or associated with the elements of an array.

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

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

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

Reduction



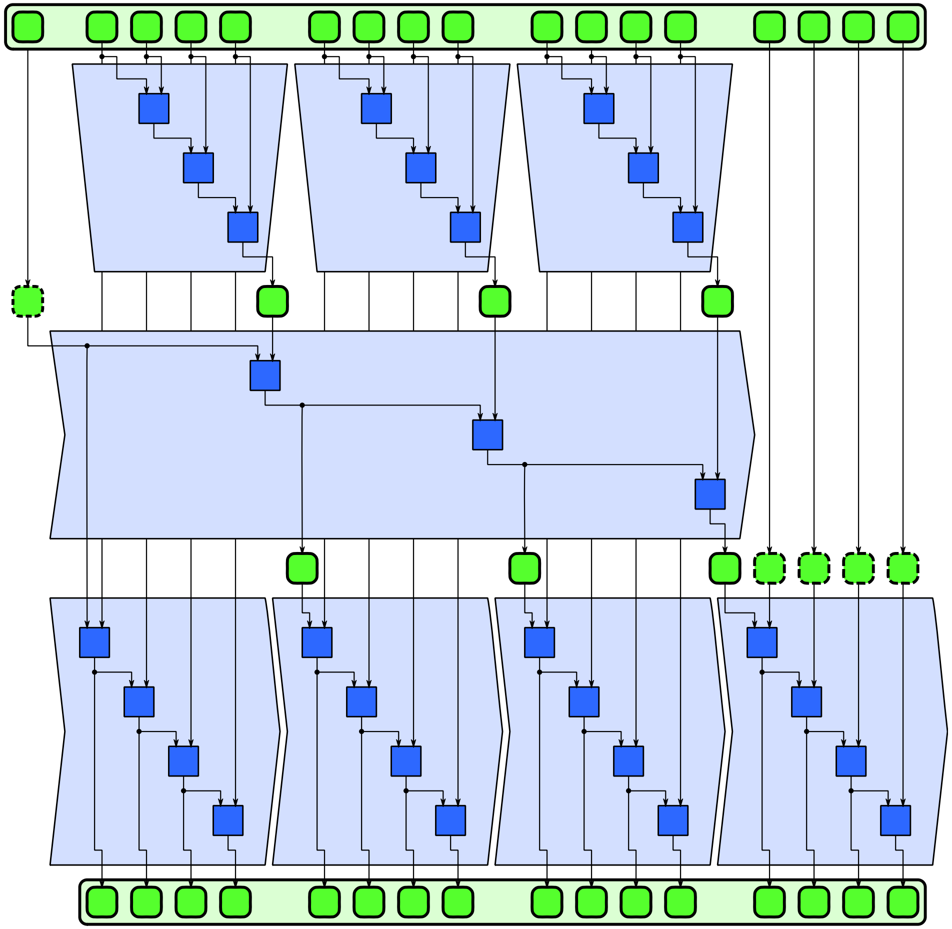
- *Reduction* combines every element in a collection into one element using an *associative* operator.

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

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

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

Scan



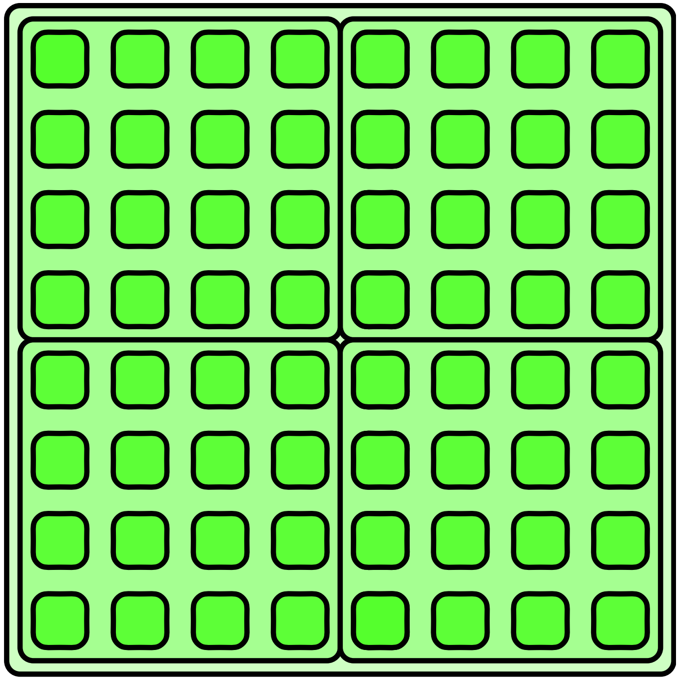
- *Scan* computes all partial reductions of a collection

```
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
- We'll consider different implementations later

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

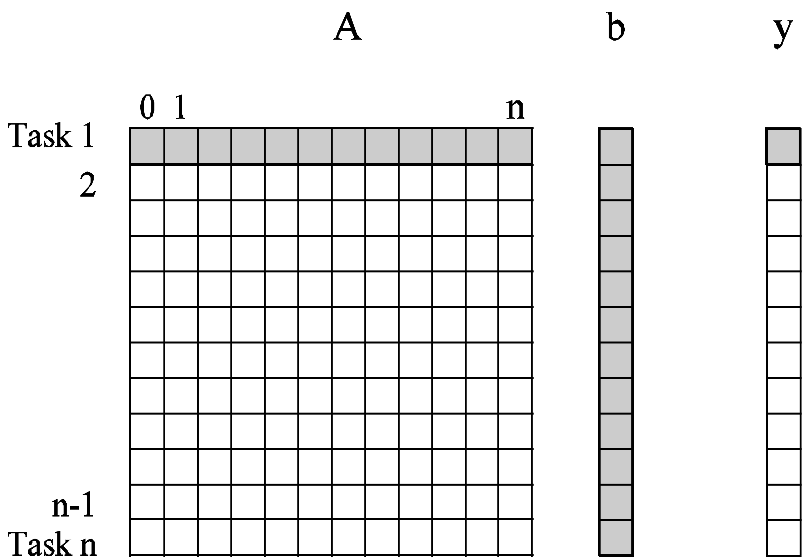
Geometric Decomposition/Partition



- *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.

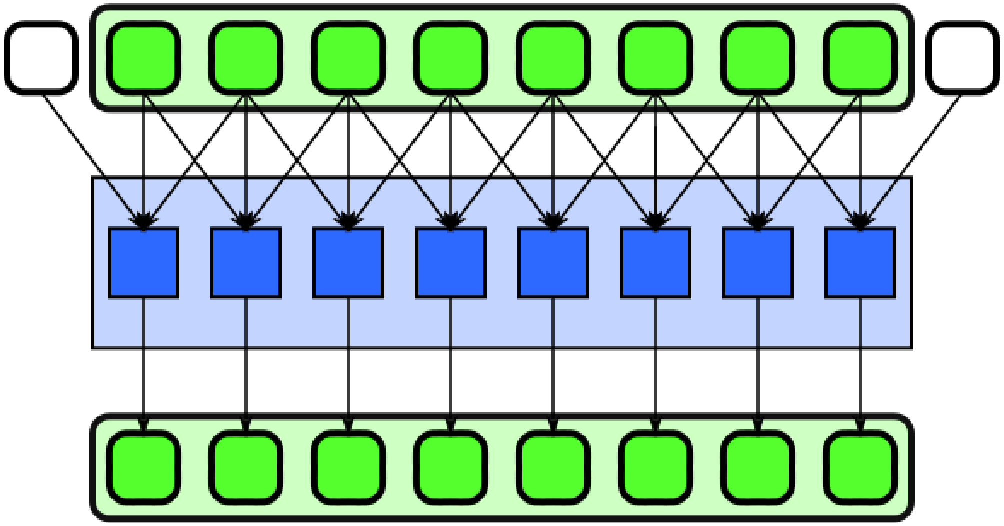
Example: Multiplying a Dense Matrix with a Vector



Computation of each element of output vector ***y*** is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into ***n*** tasks. The figure highlights the portion of the matrix and vector accessed by Task 1.

Observations: While tasks share data (namely, the vector ***b***), they do not have any control dependencies - i.e., no task needs to wait for the (partial) completion of any other. All tasks are of the same size in terms of number of operations. *Is this the maximum number of tasks we could decompose this problem into?*

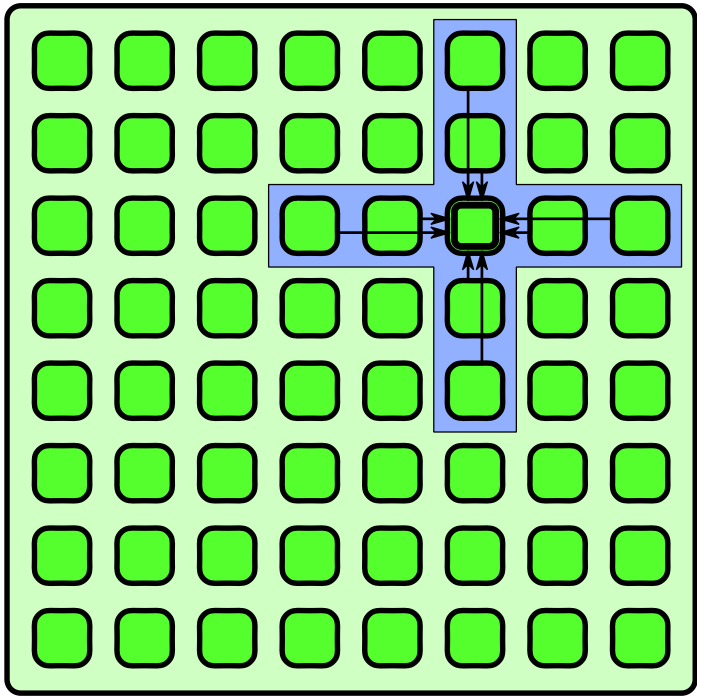
Stencil



Examples: signal filtering including convolution, median, anisotropic diffusion

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

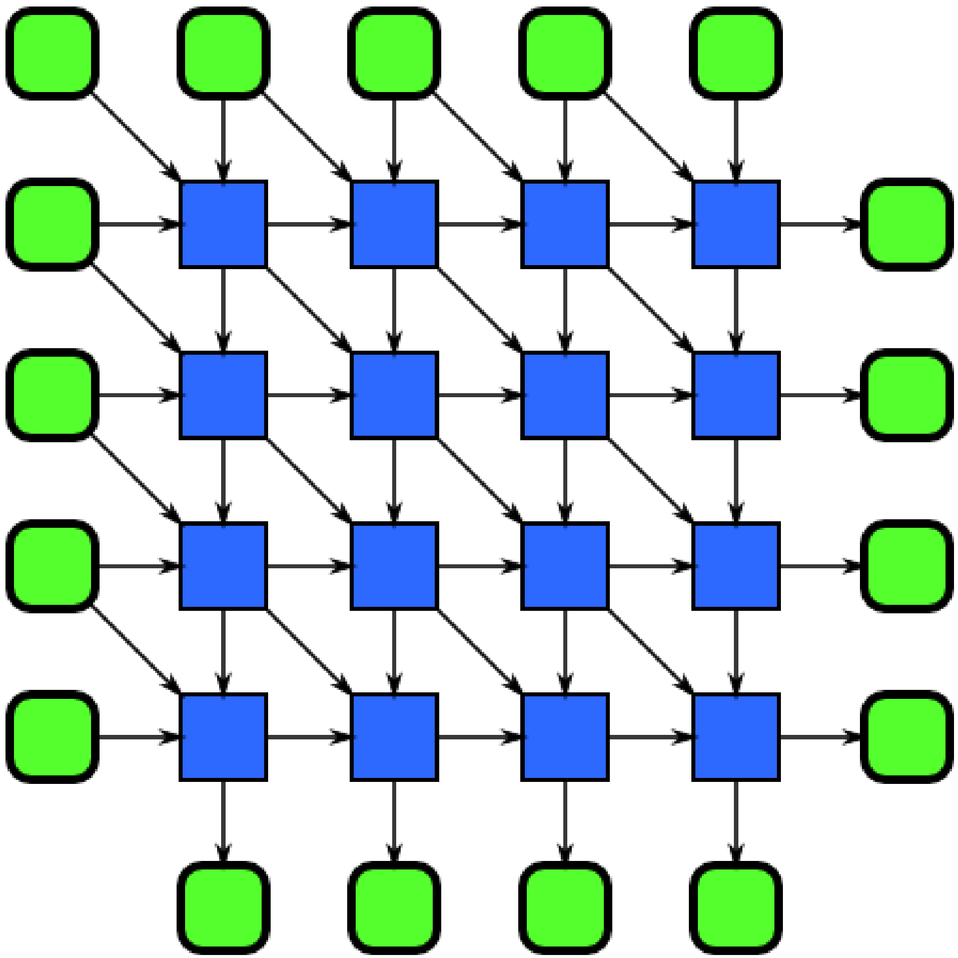
nD Stencil



- *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



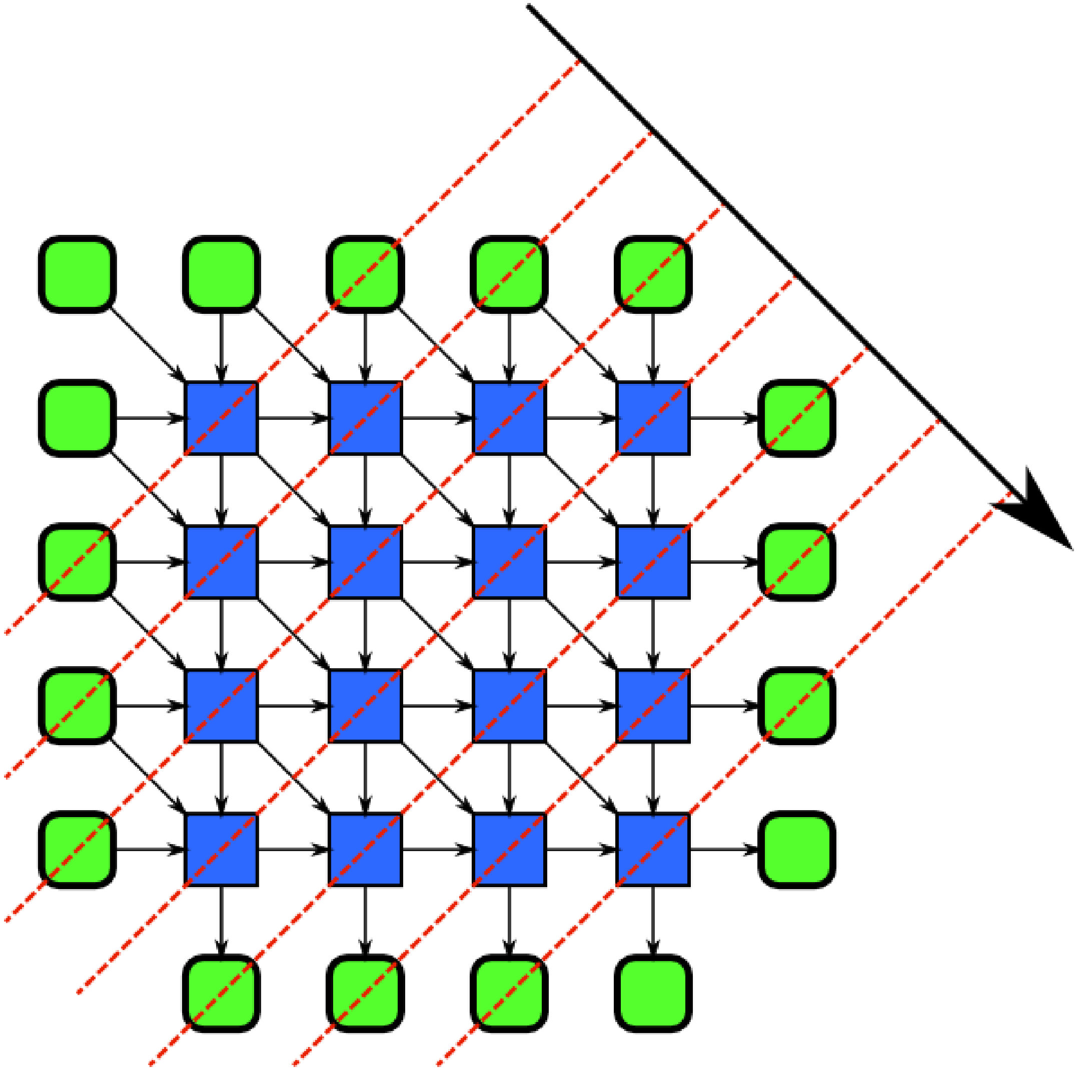
- *Recurrence* results from loop nests with both input and output dependencies between iterations
- Can also result from iterated stencils
- They must be *causal*

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

Recurrence Example

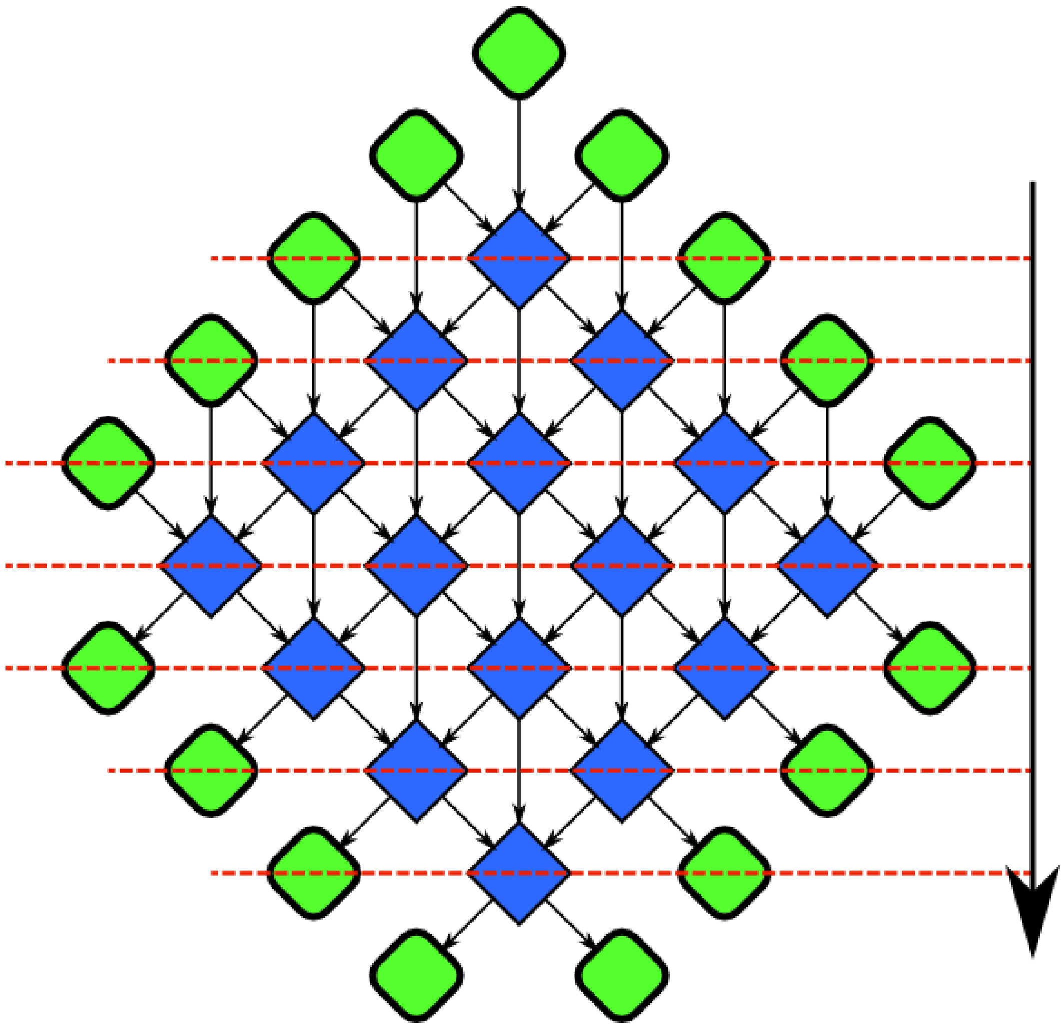
```
for (int i = 1; i < N; i++) {
  for (int j = 1; j < M; j++) {
    A[i][j] = f(
      A[i-1][j],
      A[i][j-1],
      A[i-1][j-1],
      B[i][j]);
   }
}
```

Recurrence Hyperplane Sweep



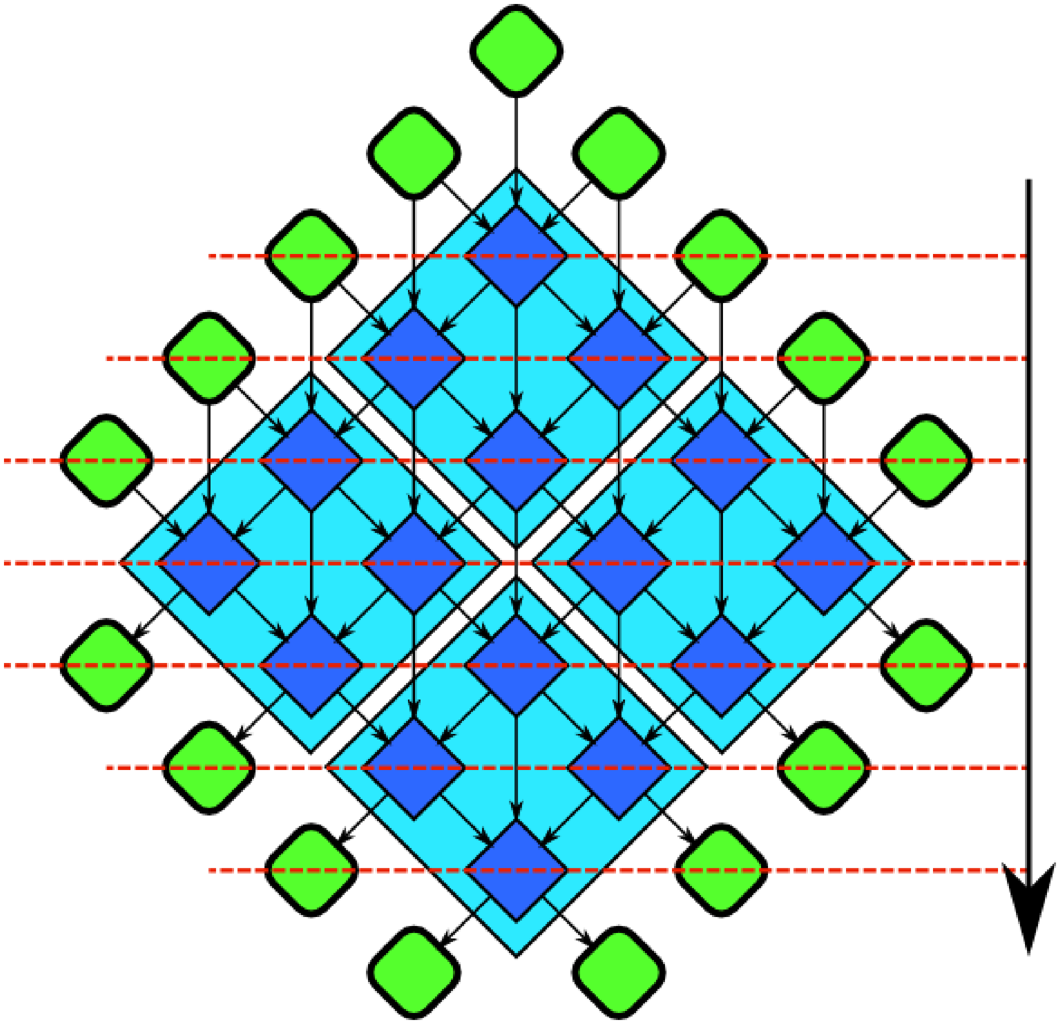
- Multidimensional recurrences can *always* be parallelized
- Leslie Lamport's hyperplane separation theorem:
  - Choose hyperplane with inputs and outputs on opposite sides
  - Sweep through data perpendicular to hyperplane

Rotated Recurrence



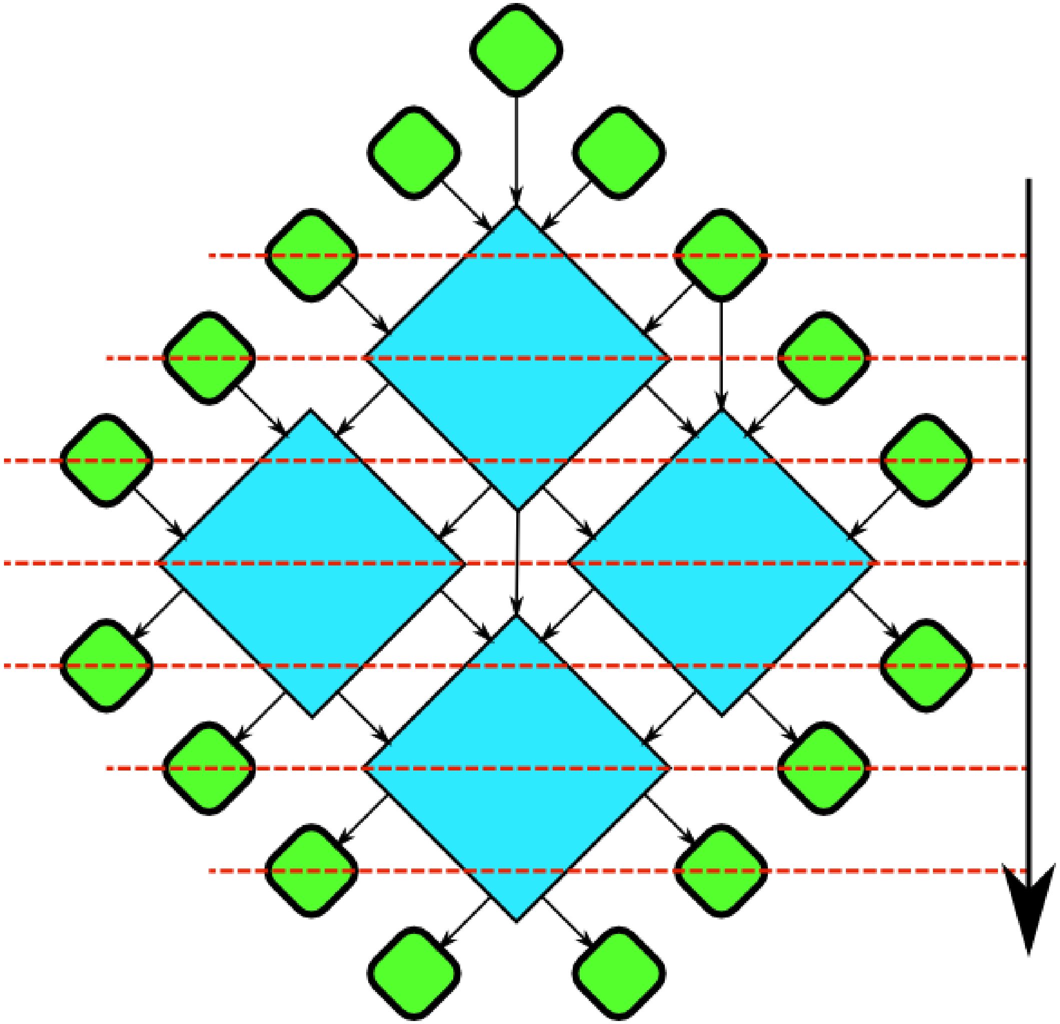
- Rotate recurrence to see sweep more clearly

Tiled Recurrence



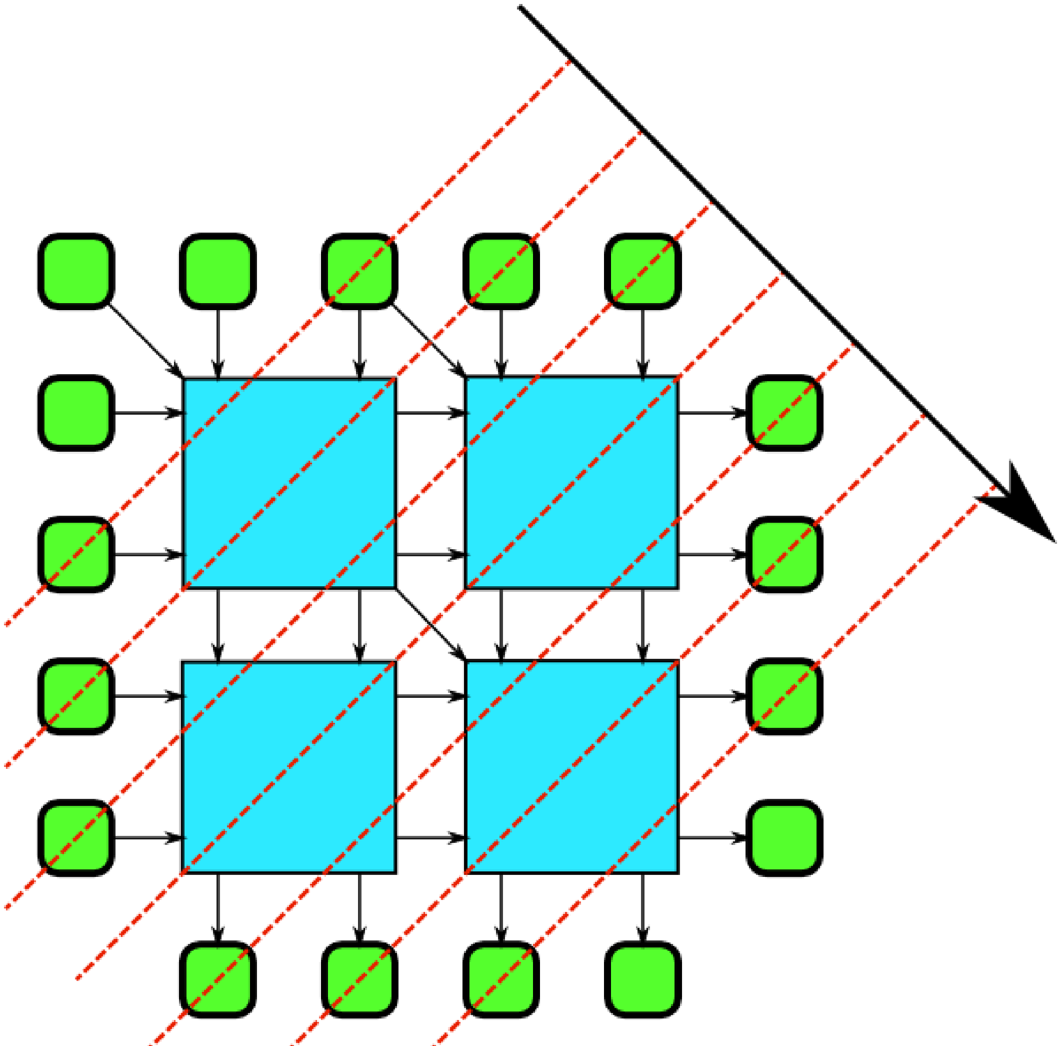
- Can partition recurrence to get a better compute vs. bandwidth ratio
- Show diamonds here, could also use paired trapezoids

Tiled Recurrence



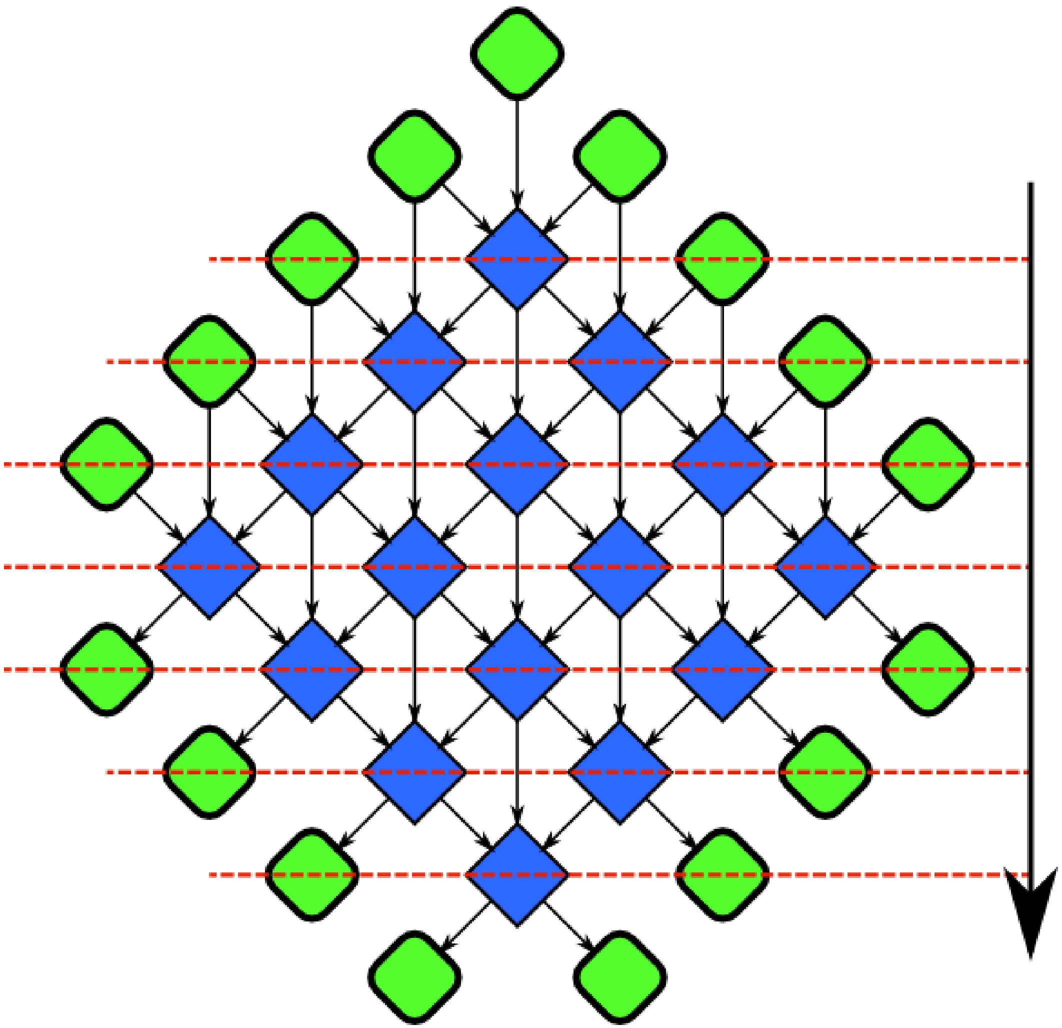
- Remove all *internalized* data dependences

Recursively Tiled Recurrences



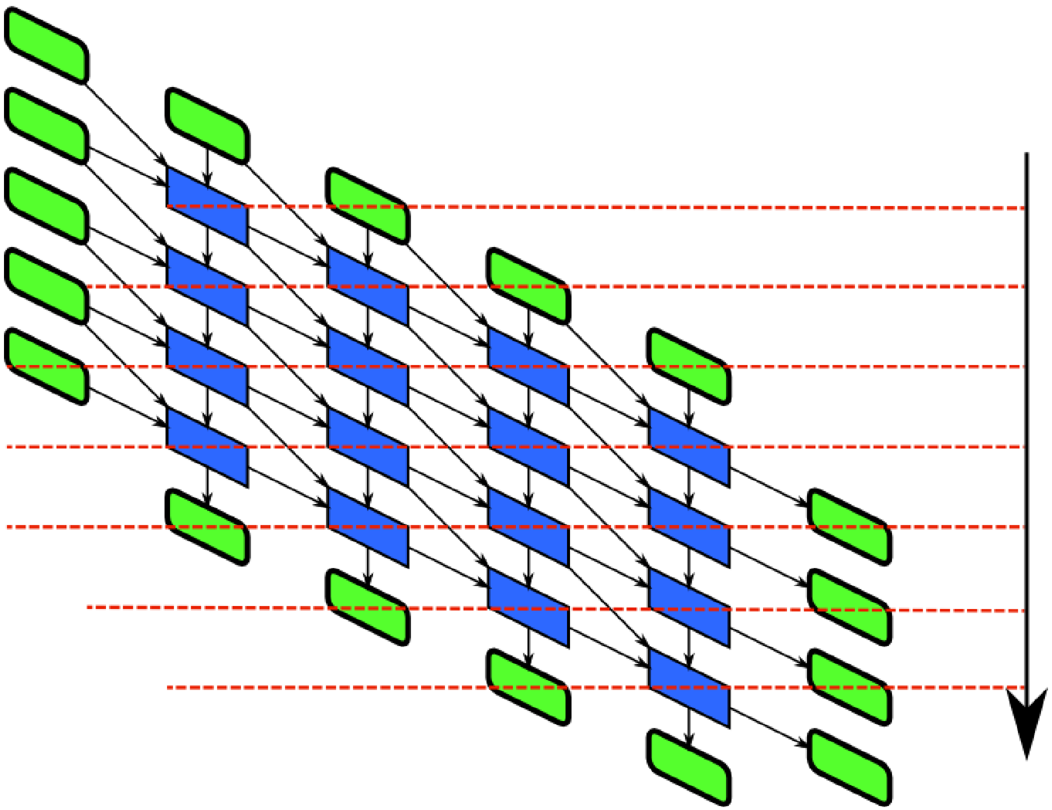
- Rotate back: same recurrence at a different scale!

Rotated Recurrence



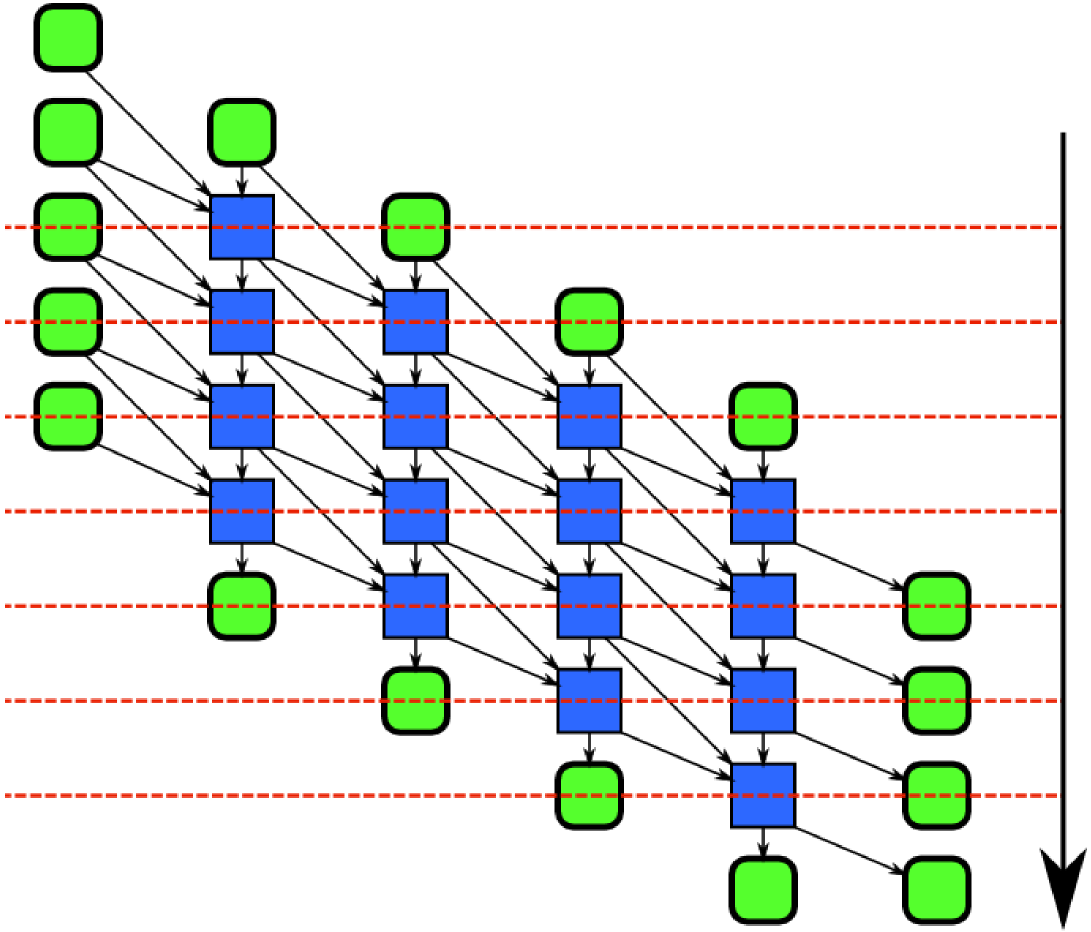
- Look at rotated recurrence again
- Let's skew this by 45 degrees…

Skewed Recurrence



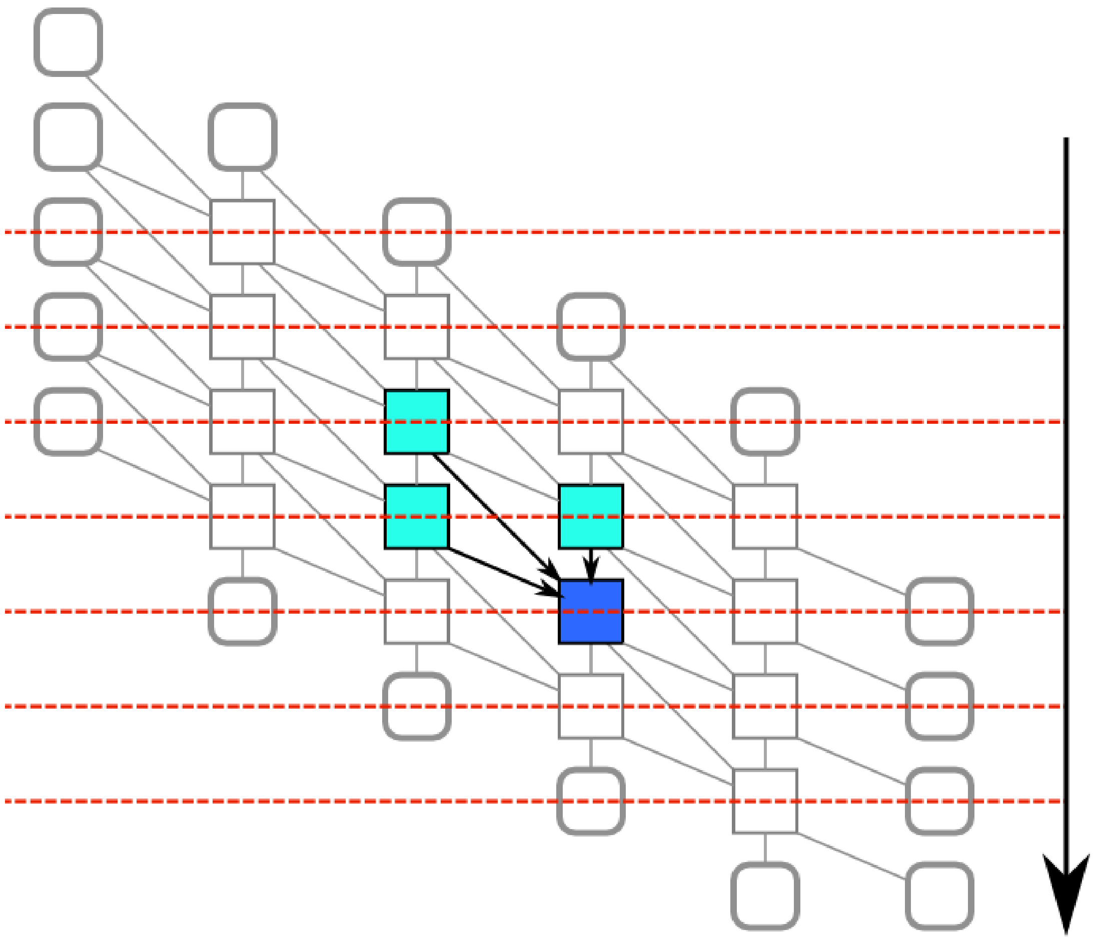
- A little hard to understand
- Let's just clean up the diagram a little bit…
    - Straighten up the symbols
    - Leave the data dependences as they are

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

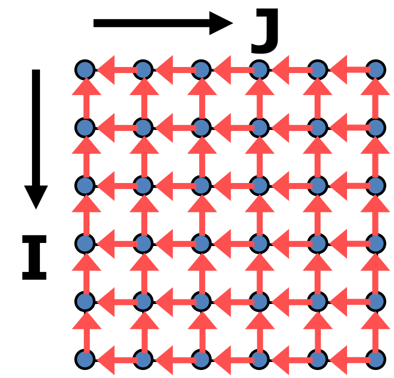


- Each element depends on certain others in previous iterations
- An iterated stencil!
- Convert iterated stencils into tiled recurrences for efficient implementation

Loop Transformations

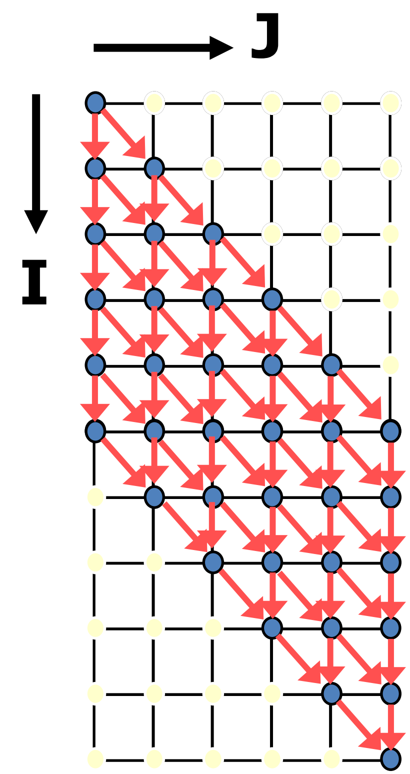
- A loop may not be parallel as is
- Example

```
FOR i = 1 to N-1
   FOR j = 1 to N-1
     A[i,j] = A[i,j-1] + A[i-1,j];
```

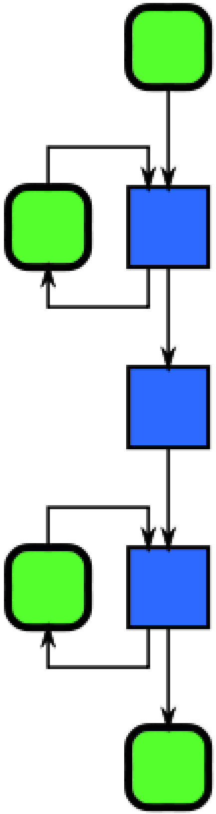


- After loop Skewing

```
FOR i = 1 to 2*N-3
   FORPAR j = max(1,i-N+2) to min(i, N-1)
     A[i-j+1,j] = A[i-j+1,j-1] + A[i-j,j];
```



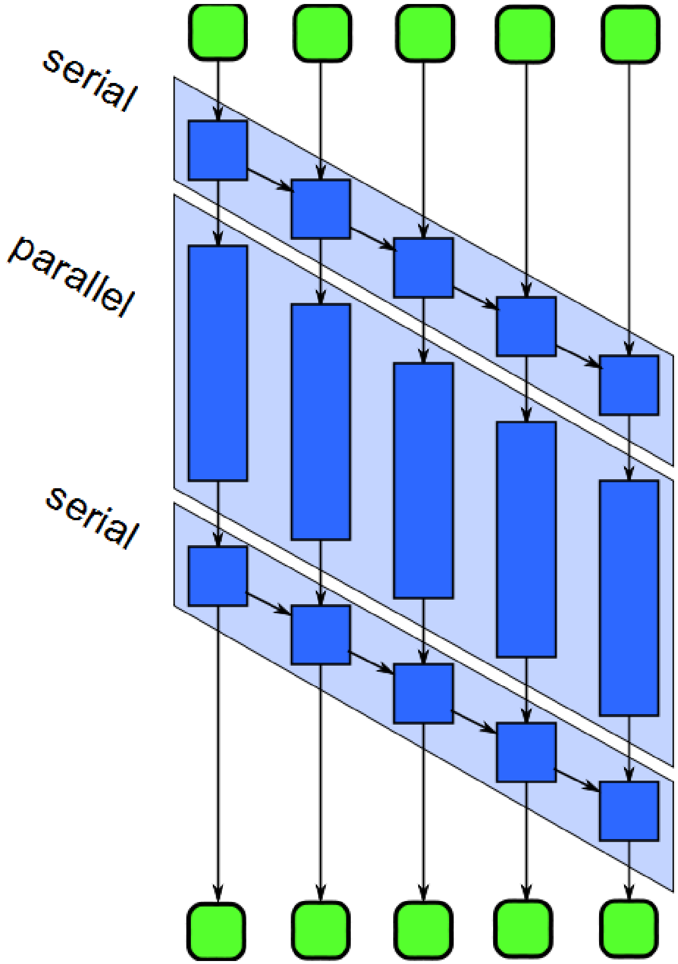
Pipeline



- *Pipeline* uses a sequence of stages that transform a flow of data
- Some stages may retain state
- Data can be consumed and produced incrementally: "online"

Examples: image filtering, data compression and decompression, signal processing

Pipeline



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

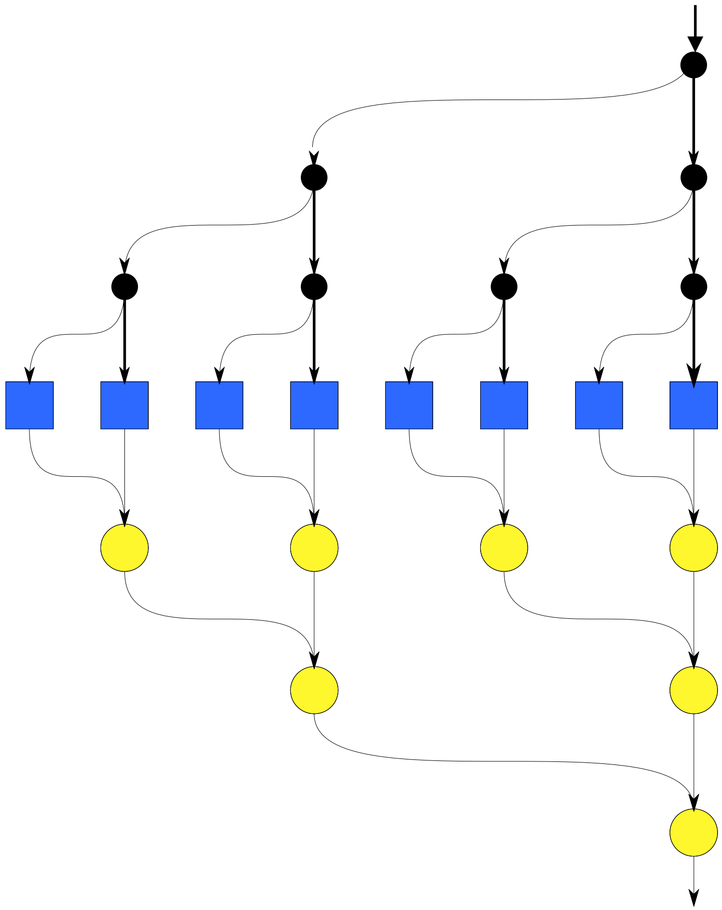
- Running multiple copies of stateless stages in parallel requires reordering of outputs

- Need to manage buffering between stages

Recursive Patterns

- Recursion is an important "universal" serial pattern
- It has roots in Lambda Calculus
  - Recursion leads to functional programming style
  - Iteration leads to procedural programming style
- Structural recursion: nesting of components
- Dynamic recursion: nesting of behaviors

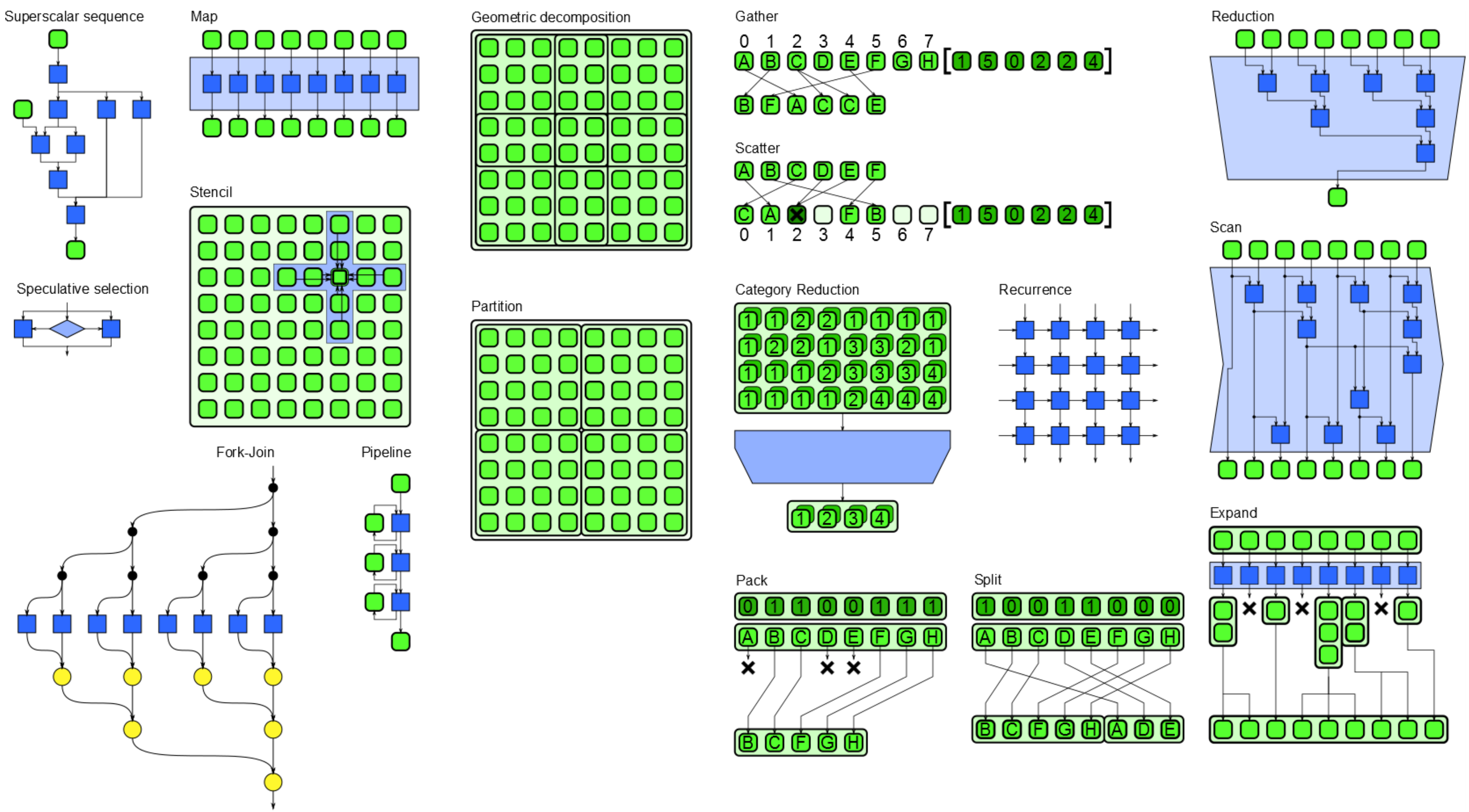
Fork-Join: Efficient Nesting



- Fork-join can be nested
- Spreads cost of work distribution and synchronization.
- This is how **cilk_for**, **tbb::parallel_for** and **arbb::map** are implemented.

Recursive fork-join enables high parallelism.

Parallel Patterns: Overview



Semantics and Implementation

Semantics: *What*

- The intended meaning as seen from the "outside"
- For example, for scan: compute all partial reductions given an associative operator

Implementation: *How*

- How it executes in practice, as seen from the "inside"
- For example, for scan: partition, serial reduction in each partition, scan of reductions, serial scan in each partition.
- *Many implementations may be possible for given semantics*
- Parallelization may require reordering of operations (associativity becomes important)
- Patterns should not over-constrain the ordering; only the important ordering constraints are specified in the semantics
- Patterns may also specify additional constraints, i.e., associativity of operators