AMS 250: An Introduction to High Performance Computing

Parallel Programming Patterns Overview

Shawfeng Dong

shaw@ucsc.edu

(831) 502-7743

Applied Mathematics & Statistics

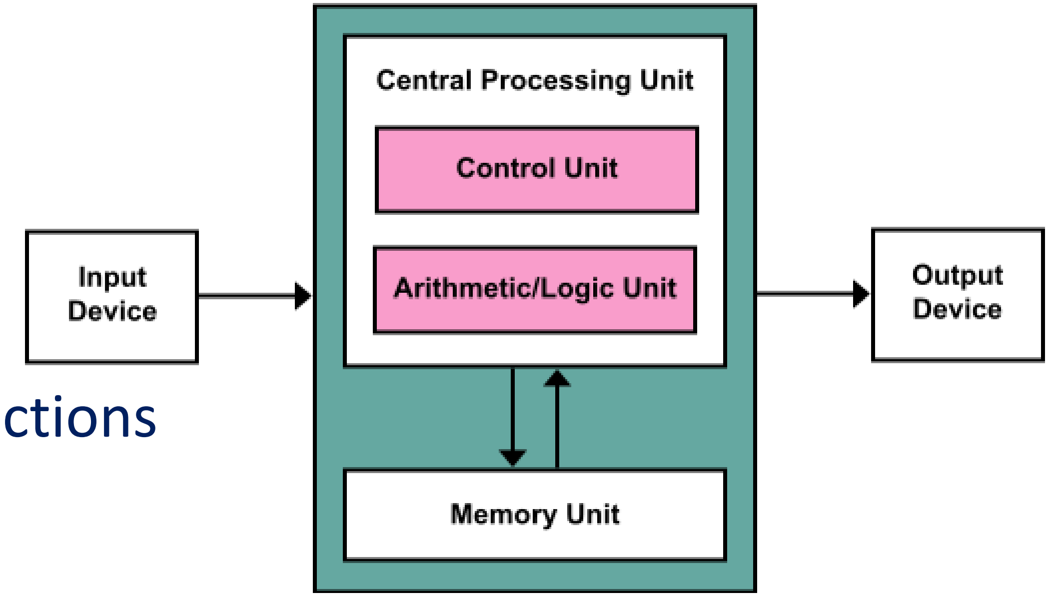
University of California, Santa Cruz

Outline

- Parallel programming models
- Dependencies
- Structured programming patterns overview

Sequential Models

- von Neumann model
  - Processing Unit, containing:
    - Arithmetic Logic Unit, with processor registers
    - Control Unit,
      with instruction registers and program counter
  - Memory, which stores both data and instructions
  - External mass storage and I/O mechanism
  - Stored-program computer
    - CPU fetches instructions from memory, reads data from memory, decodes and executes instructions sequentially, then writes data back to memory



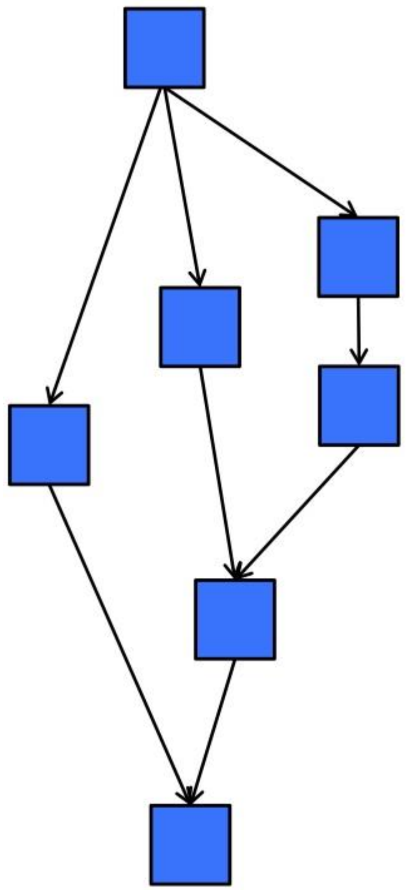
- Harvard model
  - one dedicated set of address and data buses for reading data from and writing data to memory
  - another set of address and data buses for fetching instructions

Parallel Models 101

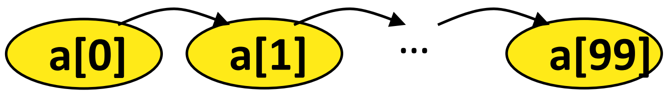
- A parallel computer is simply a collection of *processors interconnected* in some manner to *coordinate* activities and *exchange data*
- Parallel models are those theoretical models that can be used as general frameworks for describing and analyzing parallel algorithms
  - ***Simplicity***: description, analysis, architecture independence
  - ***Implementability***: able to be realized, reflect performance
- Three common parallel models
  - **Directed acyclic graphs**, **shared-memory**, **network**

Directed Acyclic Graphs (DAG)

- Captures data flow parallelism
- Nodes represent operations to be performed
  - Inputs are nodes with no incoming arcs
  - Output are nodes with no outgoing arcs
  - Think of nodes as tasks
- Arcs are paths for flow of data results
- DAG represents the operations of the algorithm and implies precedent constraints on their order

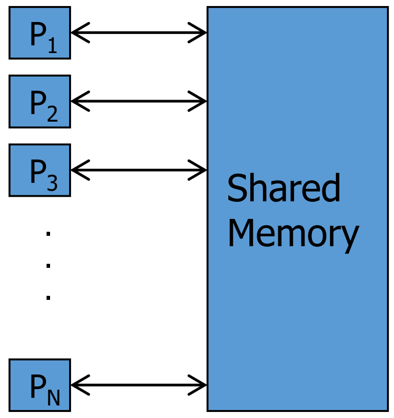


```
for (i=1; i<100; i++)
    a[i] = a[i-1] + 100;
```



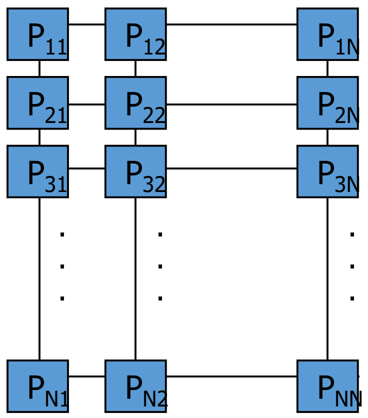
Shared Memory Model

- Parallel extension of RAM model (PRAM)
  - Memory size is infinite
  - Number of processors is unbounded
  - Processors communicate via the memory
  - Every processor accesses any memory location in 1 cycle
  - Synchronous
    - All processors execute same algorithm synchronously
      - READ phase
      - COMPUTE phase
      - WRITE phase
    - Some subset of the processors can stay idle
  - Asynchronous



Network Model

- G = (N,E)
  - **N** are processing nodes
  - **E** are bidirectional communication links
- Each processor node has its own memory
- No shared memory is available
- Network operation may be synchronous or asynchronous
- Requires communication primitives
  - Send (X, i)
  - Receive (Y, j)
- Captures **message passing** model for algorithm design



Parallelism

- Ability to execute different parts of a computation concurrently on multiple processing elements
- Why do you want parallelism?
  - Shorter running time or handling more work
- What is being parallelized?
  - *Task*: instruction, statement, procedure, …
  - *Data*: data flow, size, replication
  - Parallelism granularity
    - Coarse-grained versus fine-grained
- Thinking about parallelism
- Evaluation

Parallel Algorithm

- Recipe to solve a problem "in parallel" on multiple processing elements
- Standard steps for constructing a parallel algorithm
  - Identify work that can be performed concurrently
  - Partition the concurrent work on separate processors
  - Properly manage input, output, and intermediate data
  - Coordinate data accesses and work to satisfy dependencies

Parallelism Views

- Where can we find parallelism?
- Program (task) view
  - Statement level
    - Between program statements
    - Which statements can be executed at the same time?
  - Block level / Loop level / Routine level / Process level
    - Larger-grained program statements
- Data view
  - How is data operated on?
  - Where does data reside?
- Resource view

Parallelism, Correctness, and Dependence

- Parallel execution, from any point of view, will be constrained by the sequence of operations needed to be performed for a correct result
- Parallel execution must address control, data, and system dependences
- A *dependency* arises when one operation depends on an earlier operation to complete and produce a result before this later operation can be performed
- We extend this notion of dependency to resources since some operations may depend on certain resources
  - For example, due to where data is located

Executing Two Statements in Parallel

- Want to execute two statements in parallel
- On one processor:

  statement 1;
  statement 2;

- On two processors:

Processor 1:	Processor 2:
statement 1;	statement 2;

- Fundamental (*concurrent*) execution assumption
  - Processors execute independently of each other
  - No assumption made about speed of processor execution

Sequential Consistency in Parallel Execution

- Case 1:

Processor 1:	Processor 2:	time ↓
statement 1;		
	statement 2;	

- Case 2:

Processor 1:	Processor 2:	time ↓
	statement 2;	
statement 1;		

- Sequential consistency
  - Statements execution does not interfere with each other
  - Computation results are the same (independent of order)

Independent versus Dependent

- In other words the execution of

```
statement1;
statement2;
```

  must be equivalent to

```
statement2;
statement1;
```

- Their order of execution must not matter!
- If true, the statements are *independent* of each other
- Two statements are *dependent* when the order of their execution affects the computation outcome

Examples

- Example 1

  S1: a=1;

  S2: b=1;

  ➢ Statements are independent

- Example 2

  S1: a=1;

  S2: b=a;

  ➢ Dependent (*true (flow) dependence*)
  - Second is dependent on first
  - Can you remove dependency?

- Example 3

  S1: a=f(x);

  S2: a=b;

  ➢ Dependent (*output dependence*)
  - Second is dependent on first
  - Can you remove dependency? How?

- Example 4

  S1: a=b;

  S2: b=1;

  ➢ Dependent (*anti-dependence*)
  - First is dependent on second
  - Can you remove dependency? How?

True Dependence and Anti-Dependence

- Given statements S1 and S2,

  S1;

  S2;

- S2 has a *true (flow) dependence* on S1

  if and only if

  S2 reads a value written by S1

$$\begin{array}{ll} X = & \\ \vdots & \delta \\ \quad = X & \end{array}$$

- S2 has a *anti-dependence* on S1

  if and only if

  S2 writes a value read by S1

$$\begin{array}{ll} \quad = X & \\ \vdots & \delta^{-1} \\ X = & \end{array}$$

Output Dependence

- Given statements S1 and S2,

  S1;

  S2;

- S2 has an *output dependence* on S1

  if and only if

  S2 writes a variable written by S1

$$\begin{array}{l} X \; = \\ \quad \vdots \qquad \delta^0 \\ X \; = \end{array}$$

- Anti- and output dependences are “name” dependencies
  - Are they “true” dependences?
- How can you get rid of output dependences?
  - Are there cases where you can not?

Statement Dependency Graphs

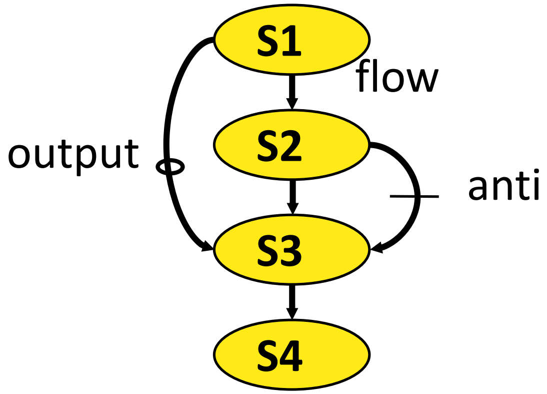
- We can use graphs to show dependence relationships
- Example

  S1: a=1;

  S2: b=a;

  S3: a=b+1;

  S4: c=a;



- $S_2 \;\; \delta \;\; S_3$: S_3 is flow-dependent on S_2
- $S_1 \;\; \delta^0 \;\; S_3$: S_3 is output-dependent on S_1
- $S_2 \;\; \delta^{-1} \; S_3$: S_3 is anti-dependent on S_2

When can two statements execute in parallel?

- Statements S1 and S2 can execute in parallel if and only if there are *no dependences* between S1 and S2
  - True dependences
  - Anti-dependences
  - Output dependences
- Some dependences can be remove by modifying the program
  - Rearranging statements
  - Eliminating statements

How do you compute dependence?

- Data dependence relations can be found by comparing the IN and OUT sets of each node
- The IN and OUT sets of a statement S are defined as:
  - IN(S) : set of memory locations (variables) that may be used in S
  - OUT(S) : set of memory locations (variables) that may be modified by S
- Note that these sets include all memory locations that may be fetched or modified
- As such, the sets can be conservatively large

IN / OUT Sets and Computing Dependence

- Assuming that there is a path from S1 to S2 , the following shows how to intersect the IN and OUT sets to test for data dependence

$out(S_1) \cap in(S_2) \neq \varnothing$	$S_1\ \delta\ S_2$	flow dependence
$in(S_1) \cap out(S_2) \neq \varnothing$	$S_1\ \delta^{-1}\ S_2$	anti-dependence
$out(S_1) \cap out(S_2) \neq \varnothing$	$S_1\ \delta^{0} S_2$	output dependence

Loop-Level Parallelism

- Significant parallelism can be identified <u>within</u> loops

```
for (i=0; i<100; i++)
  S1: a[i] = i;
```

```
for (i=0; i<100; i++) {
  S1: a[i] = i;
  S2: b[i] = 2*i;
}
```

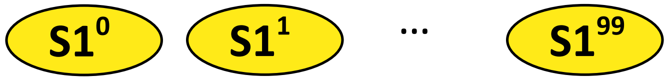
- Dependencies? What about *i*, the loop index?
- *DOALL* loop (a.k.a. *foreach* loop)
  - All iterations are independent of each other
  - All statements can be executed in parallel at the same time

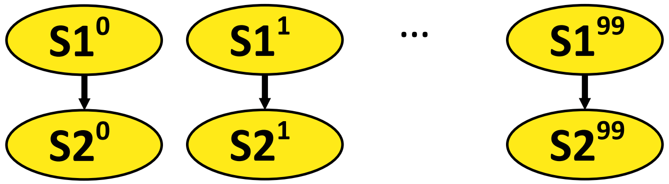
Iteration Space

- Unroll loop into separate statements / iterations
- Show dependences between iterations

```
for (i=0; i<100; i++)
  S1: a[i] = i;
```

```
for (i=0; i<100; i++) {
  S1: a[i] = i;
  S2: b[i] = 2*i;
}
```

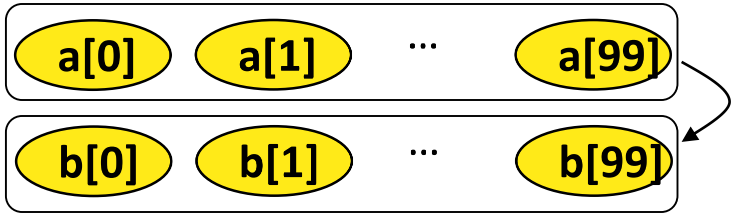




Multi-Loop Parallelism

- Significant parallelism can be identified between loops

```
for (i=0; i<100; i++) a[i] = i;
for (i=0; i<100; i++) b[i] = i;
```

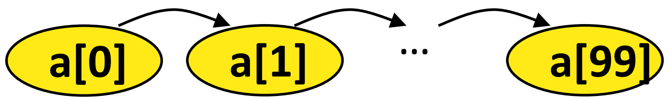


- Dependencies?
- How much parallelism is available?
- Given 4 processors, how much parallelism is possible?
- What parallelism is achievable with 50 processors?

Loops with Dependencies

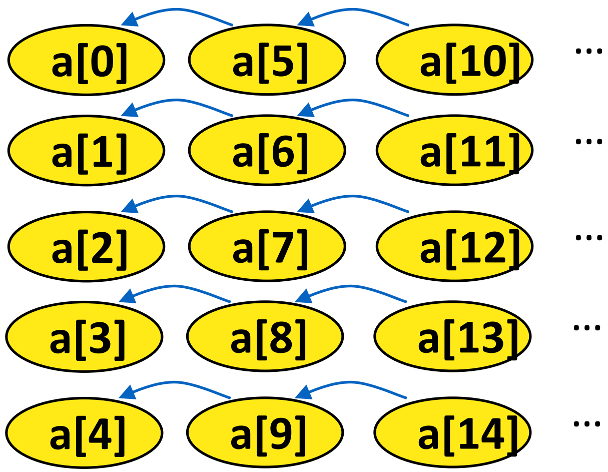
Case 1:

```
for (i=1; i<100; i++)
    a[i] = a[i-1] + 100;
```



Case 2:

```
for (i=5; i<100; i++)
    a[i-5] = a[i] + 100;
```



- Dependencies?
  - What type?
- Is the Case 1 loop parallelizable?
- Is the Case 2 loop parallelizable?

Another Loop Example

```
for (i=1; i<100; i++)
    a[i] = f(a[i-1]);
```

- Dependencies?
  - What type?
- Loop iterations are not parallelizable
  - Why not?

Loop Dependencies

- A *loop-carried* dependence is a dependence that is present only if the statements are part of the execution of a loop (i.e., between two statements instances in two different iterations of a loop)
- Otherwise, it is *loop-independent*, including between two statements instances in the same loop iteration
- Loop-carried dependences can prevent loop iteration parallelization
- The dependence is *lexically forward* if the source comes before the target or *lexically backward* otherwise
    - Unroll the loop to see

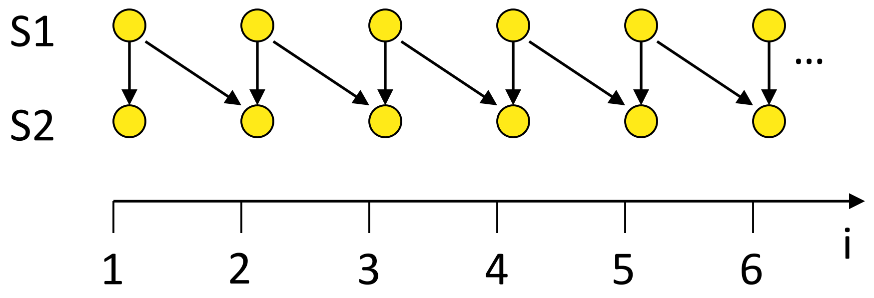
Loop Dependence Example

```
for (i=0; i<100; i++)
    a[i+10] = f(a[i]);
```

- Dependencies?
  - Between a[10], a[20], …
  - Between a[11], a[21], …
- Some parallel execution is possible
  - How much?

Dependences Between Iterations

```
for (i=1; i<100; i++) {
   s1: a[i] = …;
   s2: … = a[i-1];
}
```



- Dependencies?
  - Between a[i] and a[i-1]
- Is parallelism possible?
  - Statements can be executed in “pipeline” manner

Another Loop Dependence Example

```
for (i=0; i<100; i++)
    for (j=1; j<100; j++)
        a[i][j] = f(a[i][j-1]);
```

- Dependencies?
  - Loop-independent dependence on i
  - Loop-carried dependence on j
- Which loop can be parallelized?
  - Outer loop parallelizable
  - Inner loop cannot be parallelized

Still Another Loop Dependence Example

```
for (j=1; j<100; j++)
    for (i=0; i<100; i++)
        a[i][j] = f(a[i][j-1]);
```

- Dependencies?
  - Loop-independent dependence on i
  - Loop-carried dependence on j
- Which loop can be parallelized?
  - Inner loop parallelizable
  - Outer loop cannot be parallelized
  - Less desirable (why?)

Key Ideas for Dependency Analysis

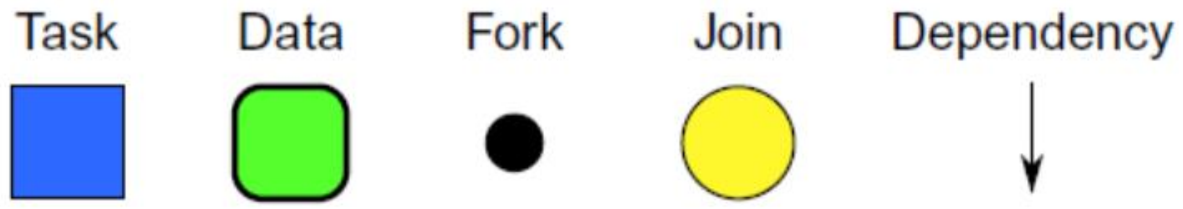
- To execute in parallel:
  - Statement order must not matter
  - Statements must not have dependences
- Some dependences can be removed
- Some dependences may not be obvious

Dependencies and Synchronization

- How is parallelism achieved when having dependencies?
  - Think about concurrency
  - Some parts of the execution are independent
  - Some parts of the execution are dependent
- Must control ordering of events on different processors (cores)
  - Dependencies pose constraints on parallel event ordering
  - Partial ordering of execution action
- Use synchronization mechanisms
  - Need for concurrent execution too
  - Maintains partial order

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.



Graphical notation for the fundamental components of algorithms

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.

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

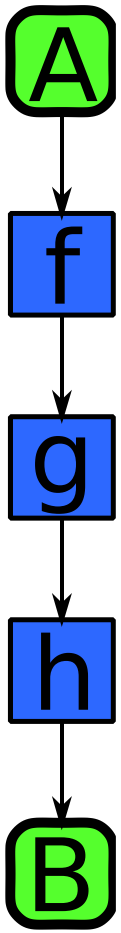
- **Serial:** Sequence

→ **Parallel:** Superscalar Sequence

- **Serial:** Iteration

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

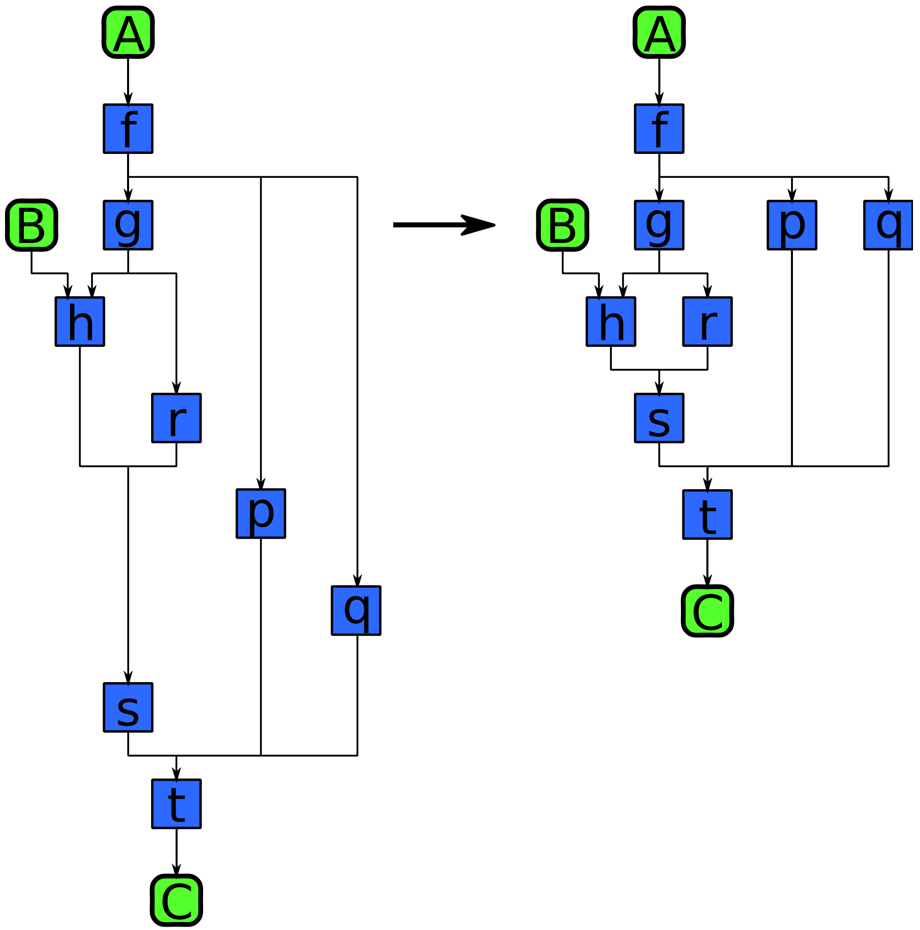
(Serial) Sequence



A serial sequence is executed in the exact order given:

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

Superscalar Sequence

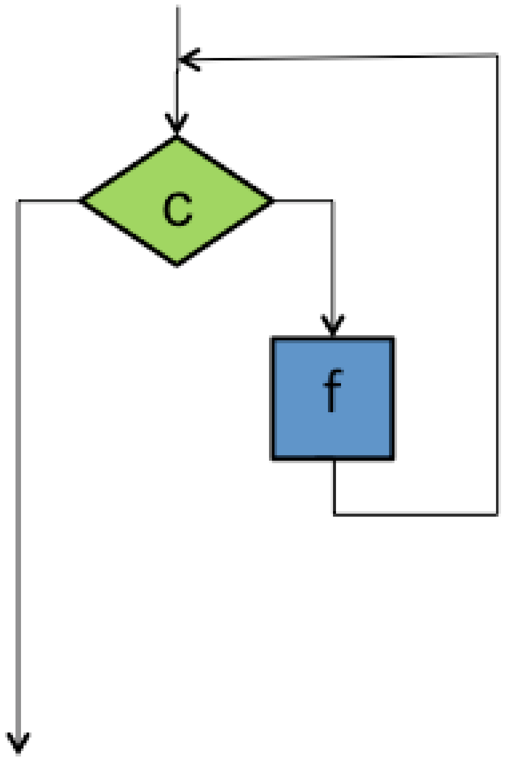


Developer writes “serial” code:

```
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



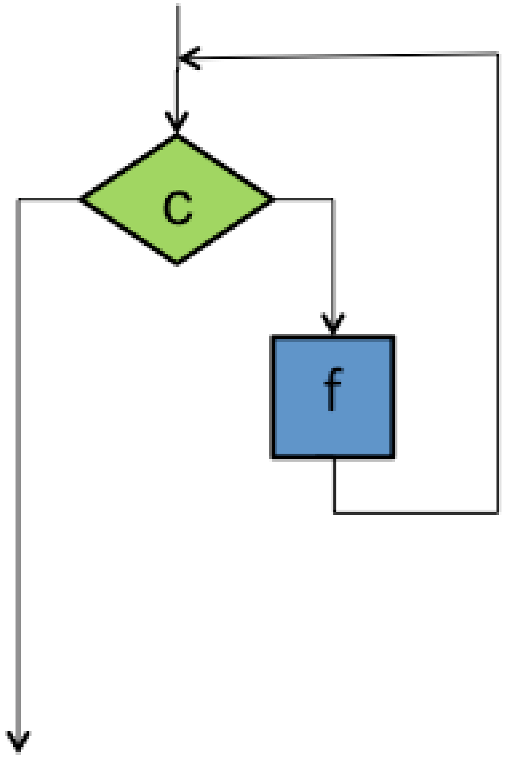
The iteration pattern repeats some section of code as long as a condition holds

```
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();
}
```

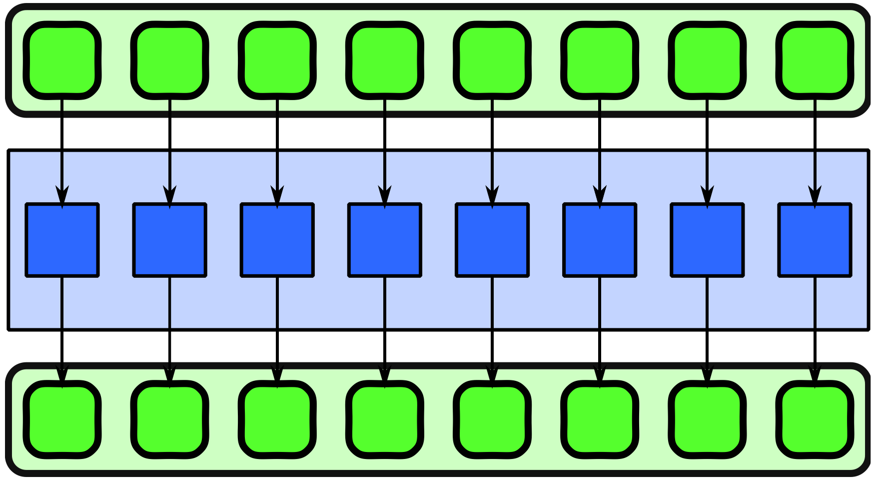
This is the same as

```
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



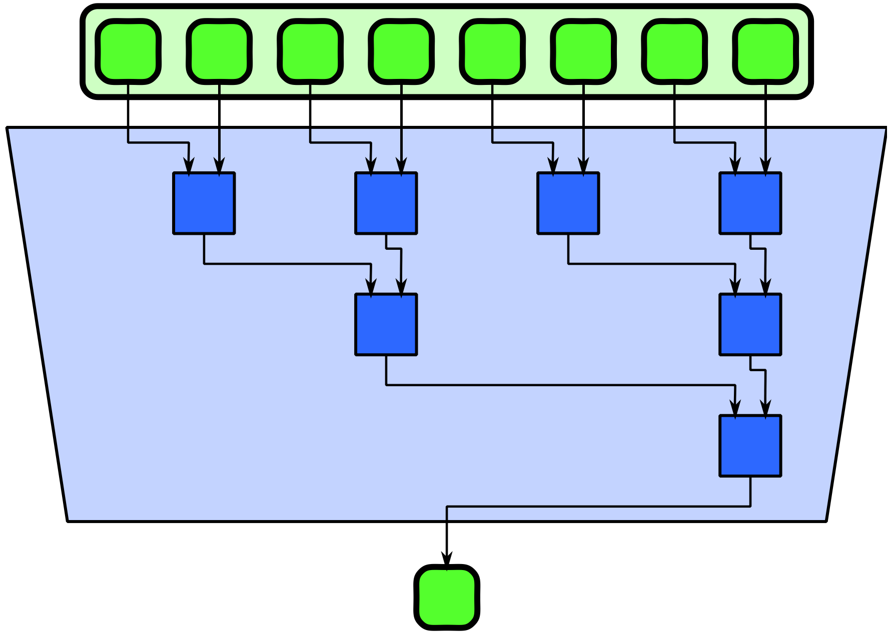
Examples: gamma correction and thresholding in images; color space conversions; Monte Carlo sampling; ray tracing.

- *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



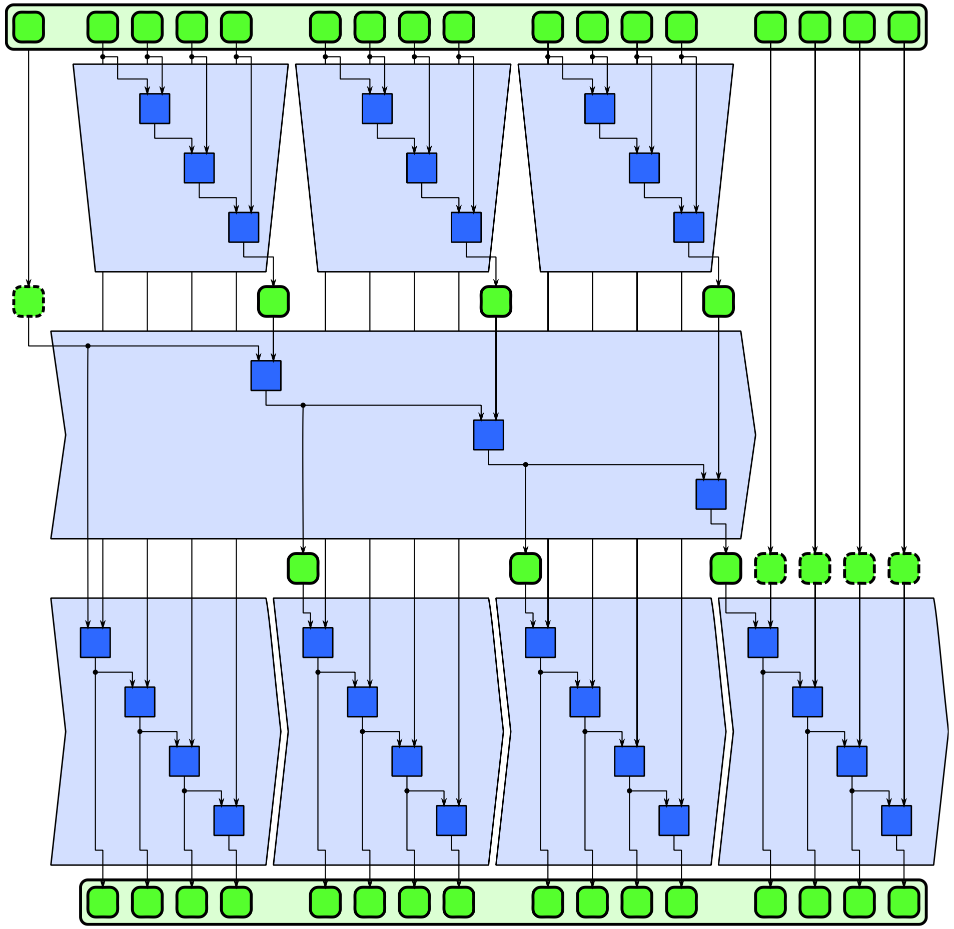
Examples: averaging of Monte Carlo samples; convergence testing; image comparison metrics; matrix operations.

- *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.

Scan



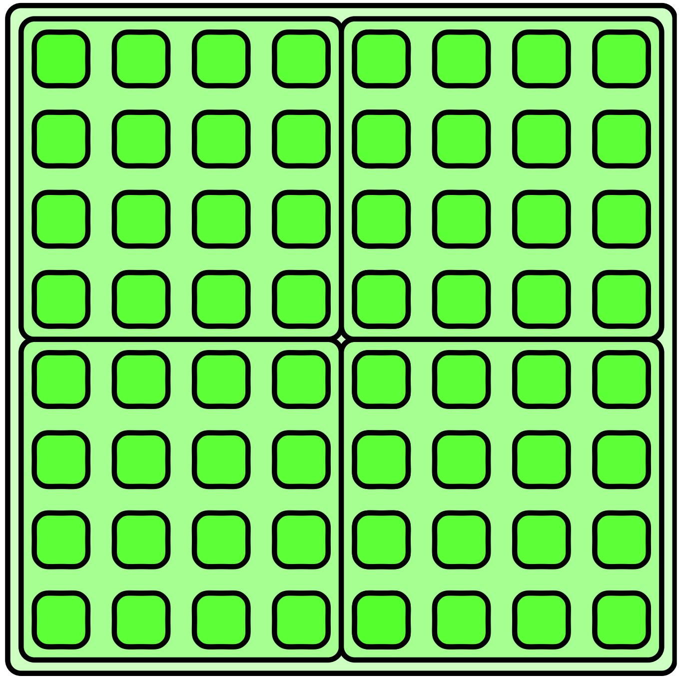
- *Scan* computes all partial reductions of a collection

```
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

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

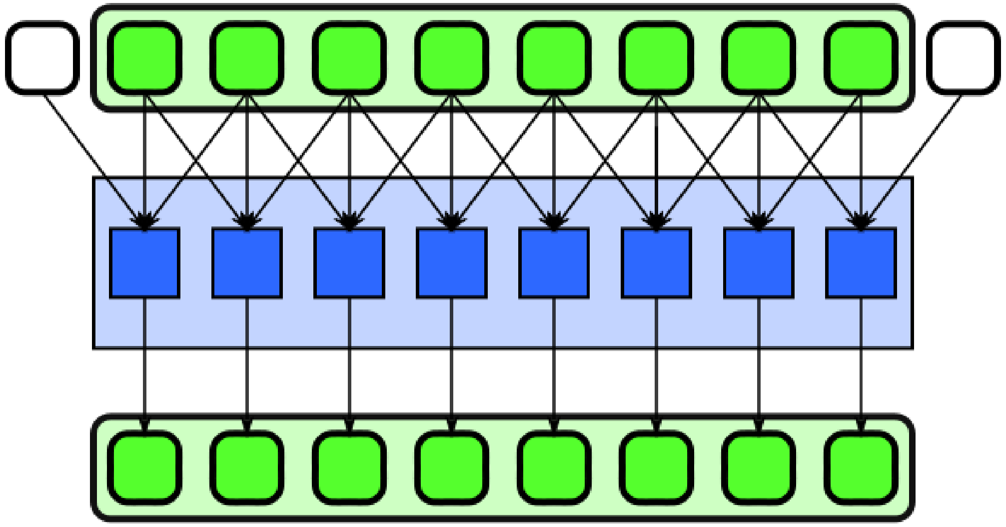
Geometric Decomposition/Partition



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

- *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

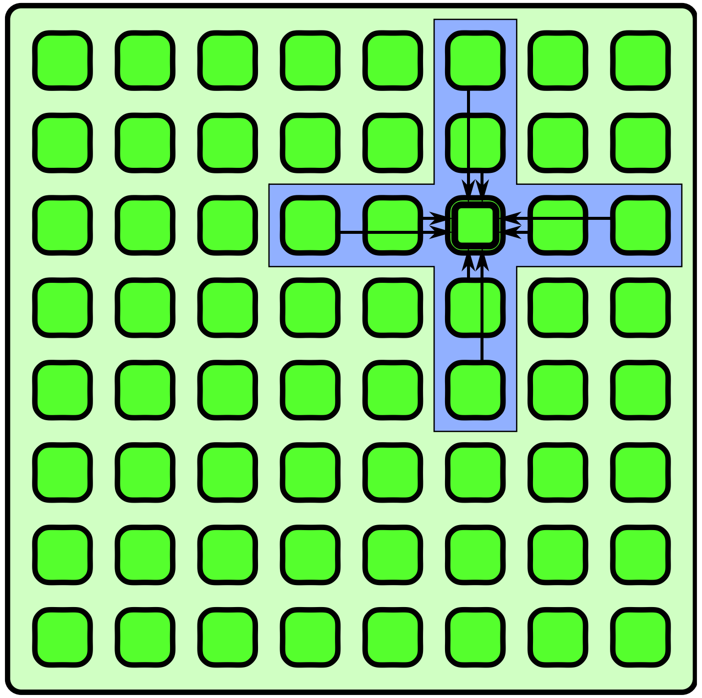
Stencil



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.

nD Stencil

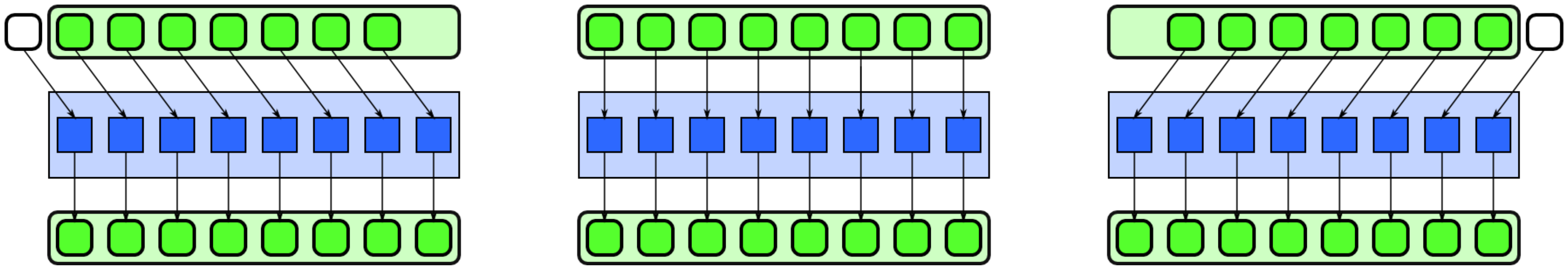


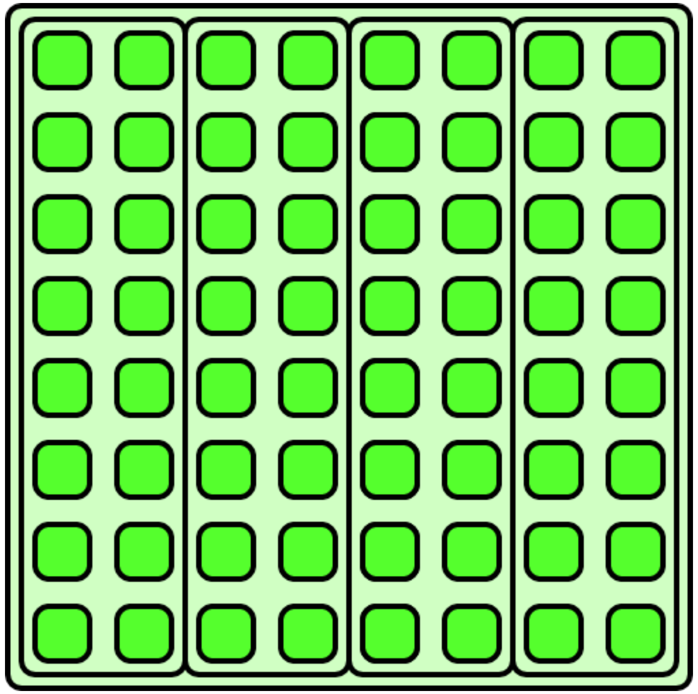
- *nD Stencil* applies a function to neighbourhoods of an nD array
- Neighbourhoods are given by set of relative offsets
- Boundary conditions need to be considered

```
for (int i = 1, i < N; i++)
  for (int j = 1, j < M; j++)
    a_new[i][j] = 0.25 *
                  (a[i-1][j] +
                   a[i+1][j] +
                   a[i][j-1] +
                   a[i][j+1]);
```

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

Implementing Stencil

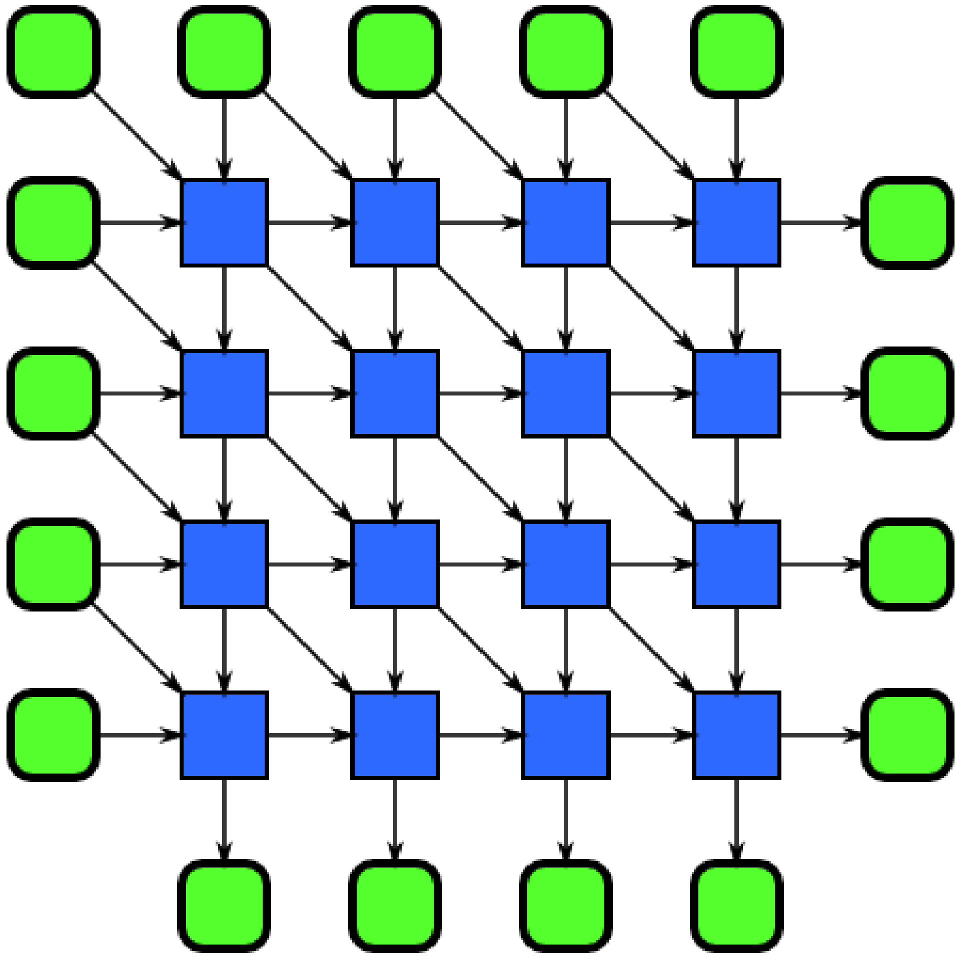




Vectorization can include converting regular reads into a set of shifts.

Strip-mining reuses previously read inputs within serialized chunks.

Recurrence

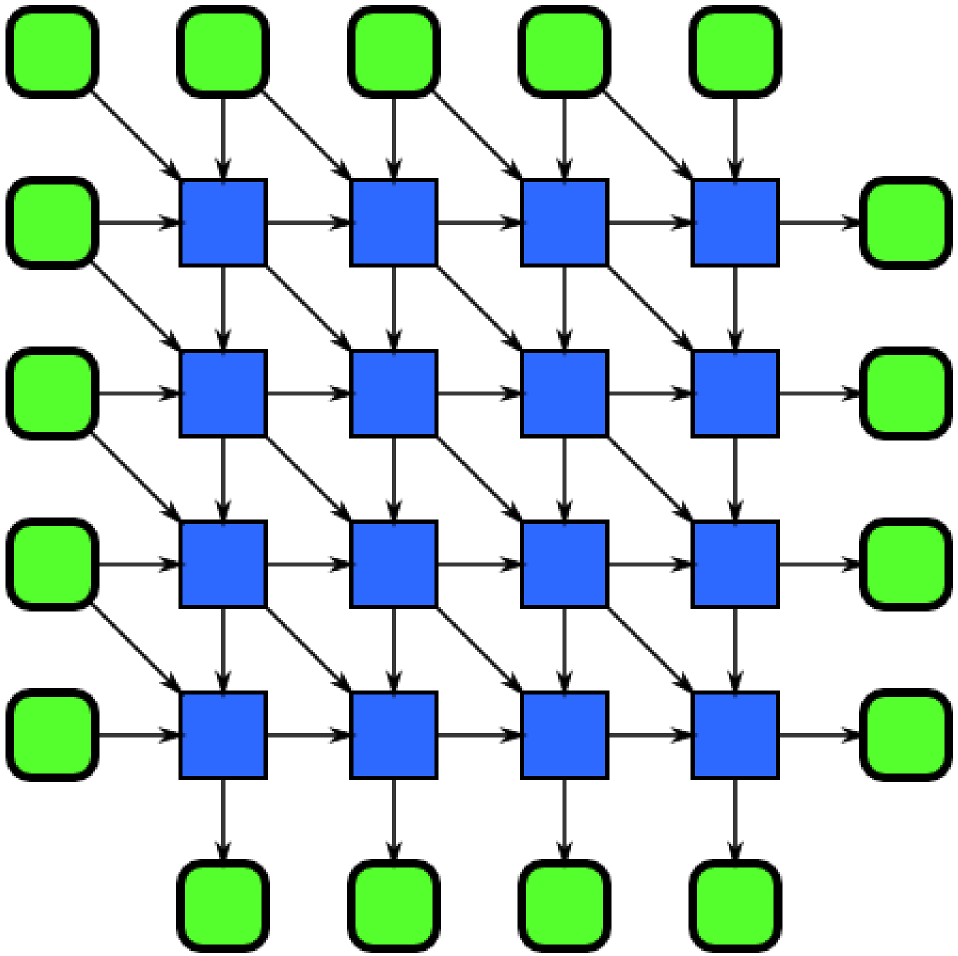


- *Recurrence* results from loop nests with both input and output dependencies between iterations
- Can also result from iterated stencils

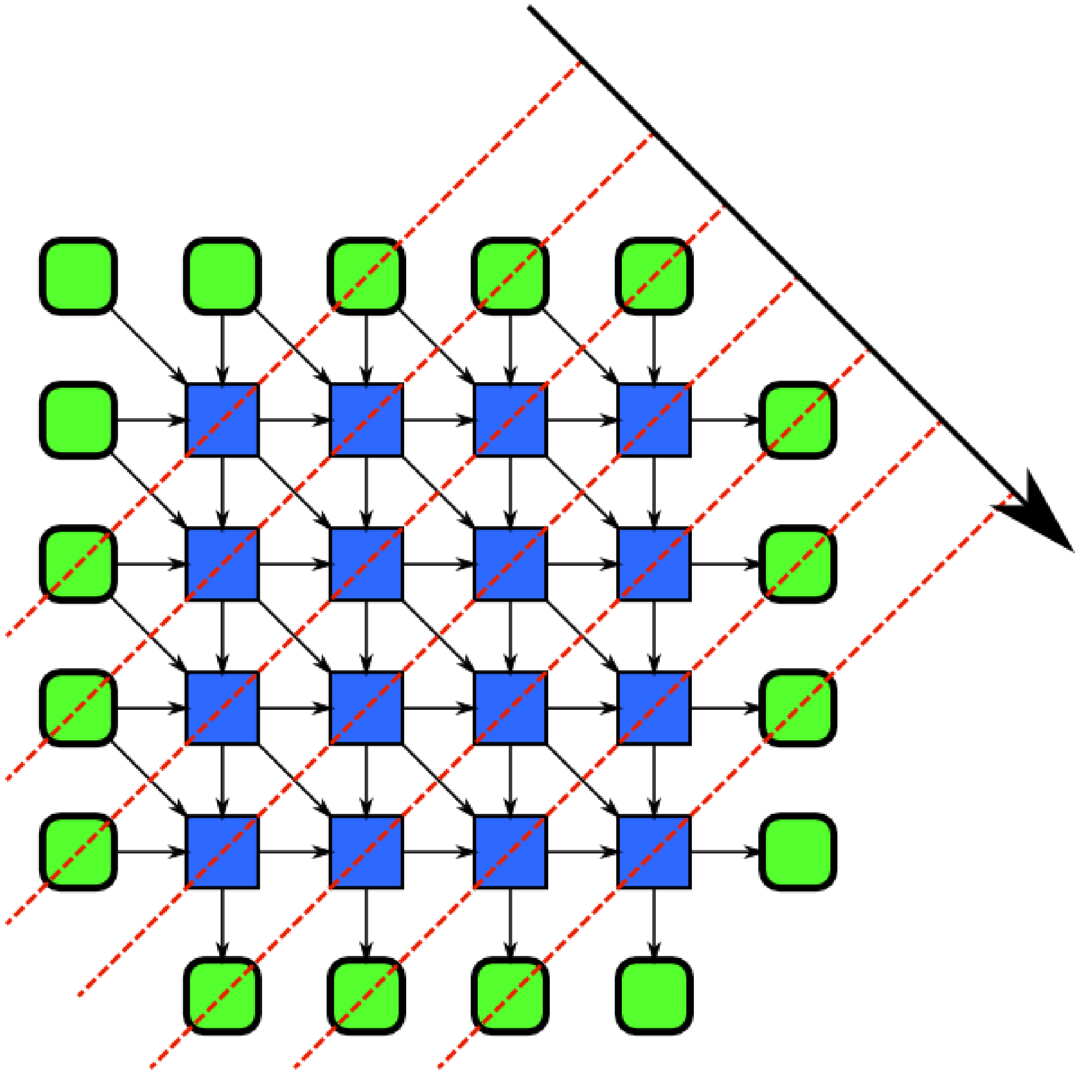
Examples: Simulation including fluid flow, electromagnetic, and financial PDE solvers, lattice QCD, sequence alignment and pattern matching

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],
      A[i][j]);
  }
}
```

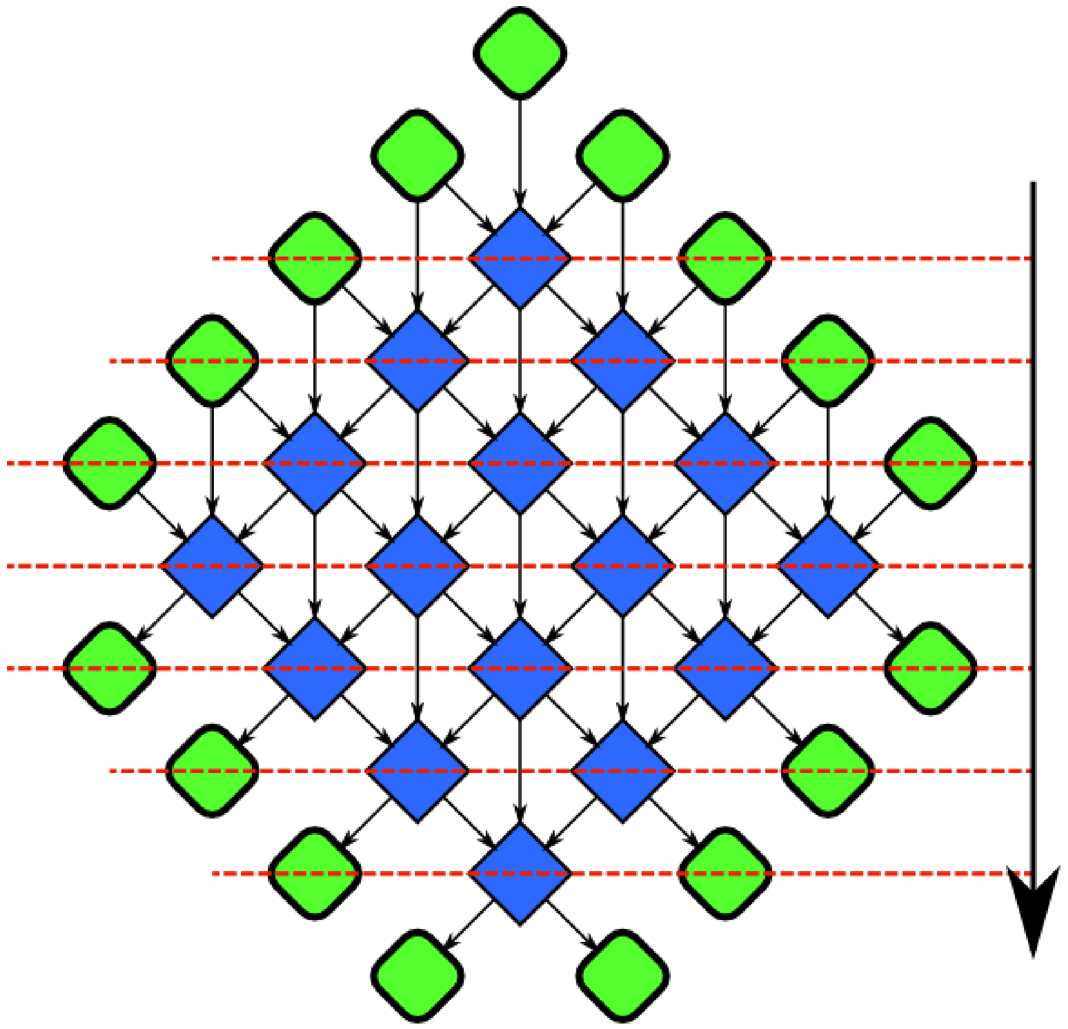


Recurrence Hyperplane Sweep



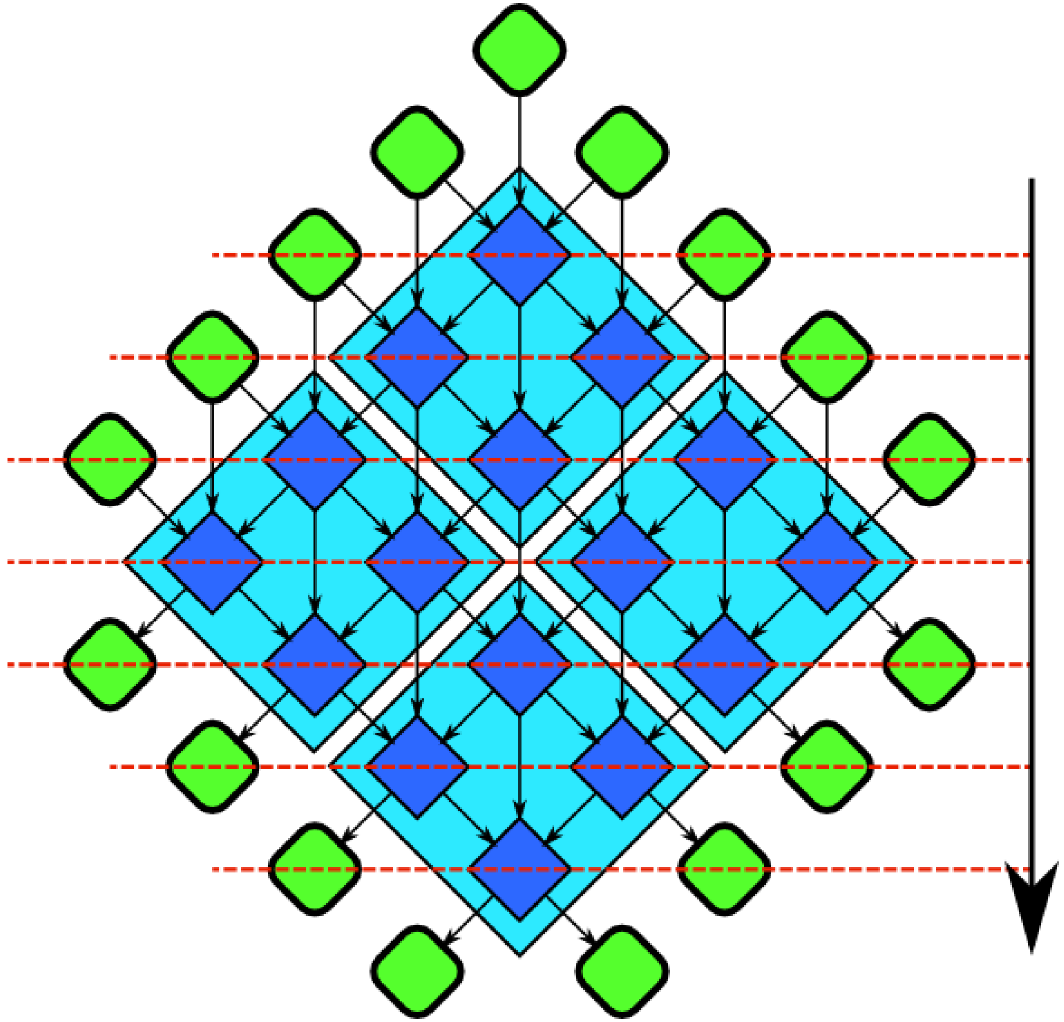
- Multidimensional recurrences can *always* be parallelized
- Leslie Lamport's hyperplane separation theorem:
  - Choose hyperplane with inputs and outputs on opposite sides
  - Sweep through data perpendicular to hyperplane

Rotated Recurrence



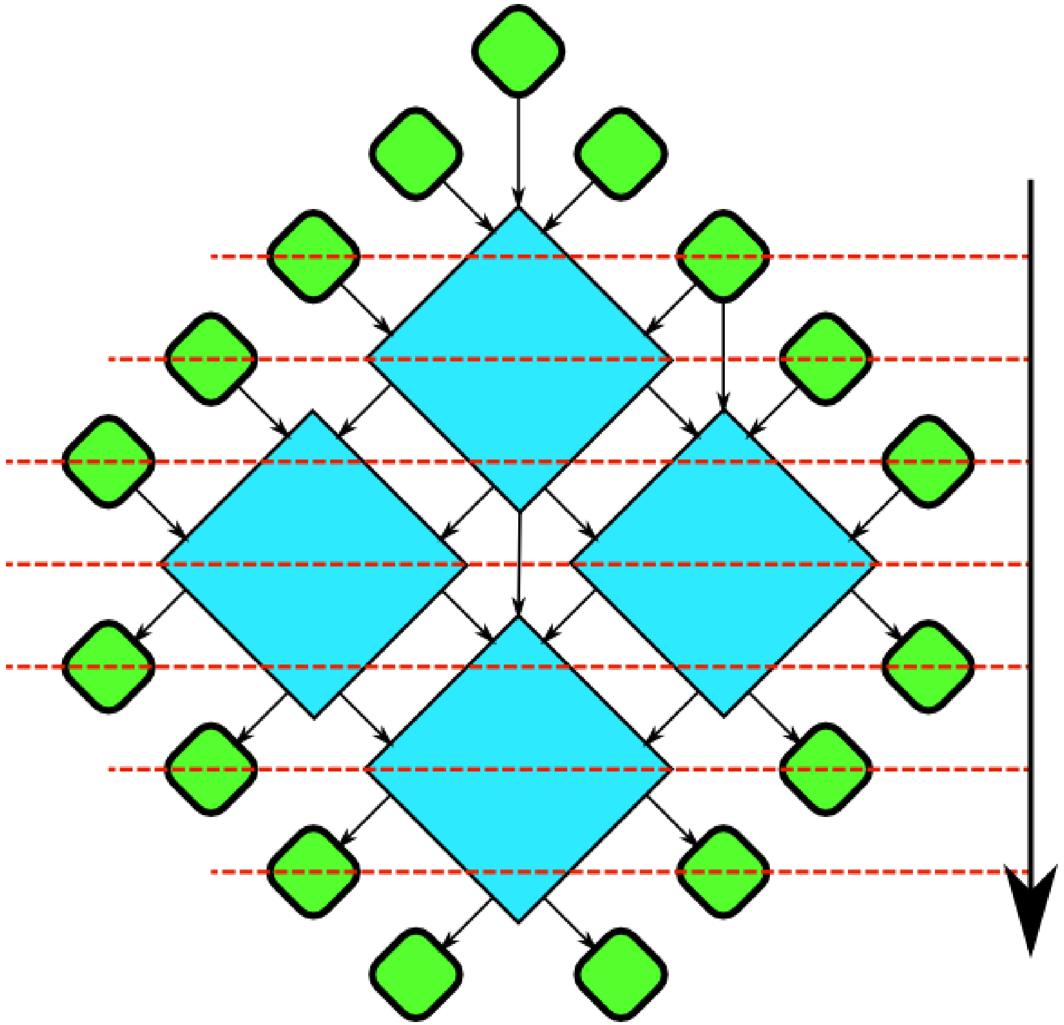
- Rotate recurrence to see sweep more clearly

Tiled Recurrence



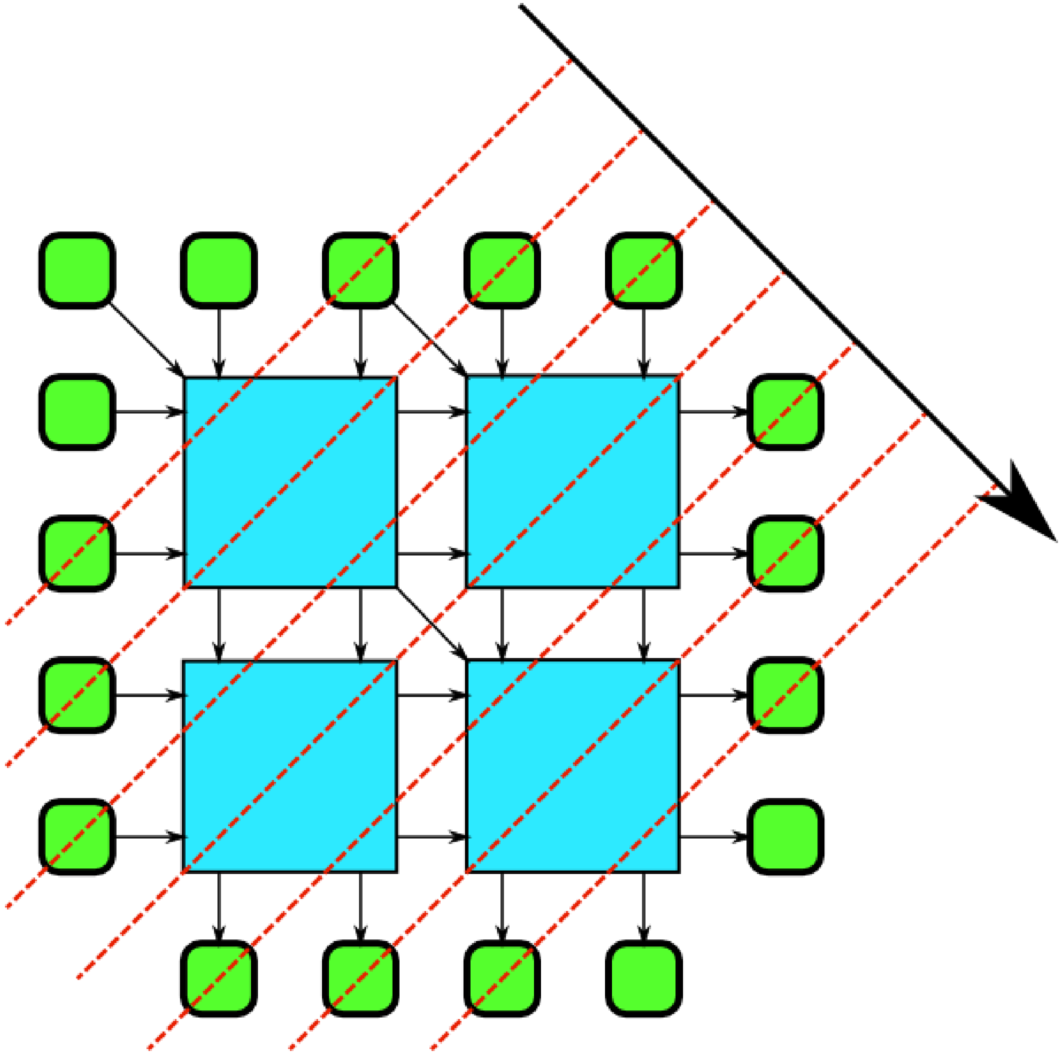
- Can partition recurrence to get a better compute vs. bandwidth ratio

Tiled Recurrence



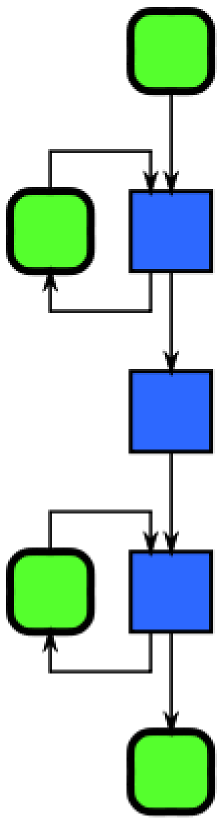
- Remove all non-redundant data dependences

Recursively Tiled Recurrences



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

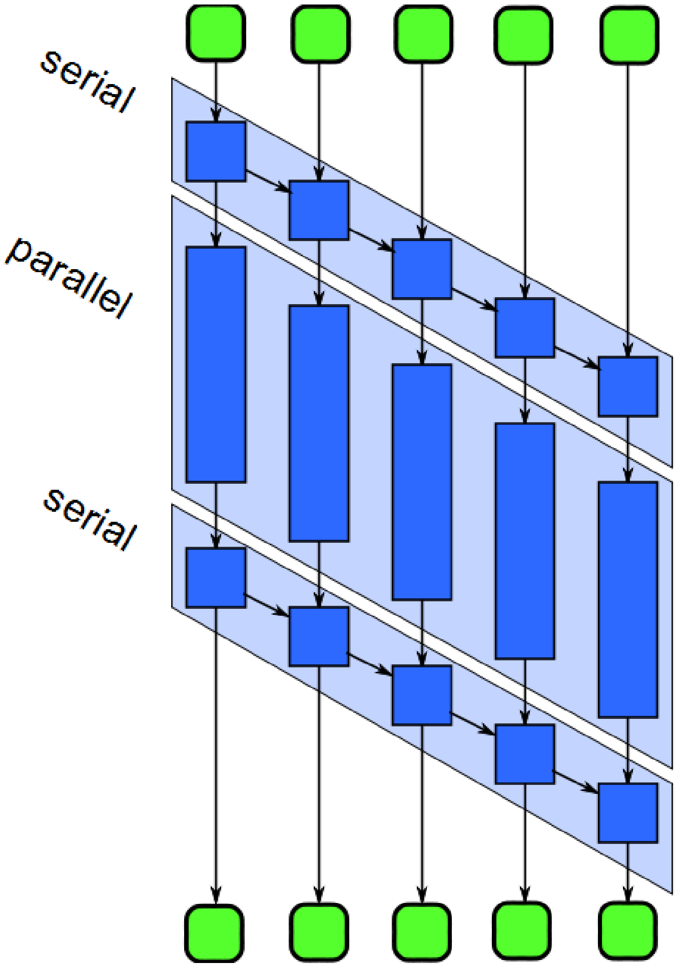
Pipeline



- *Pipeline* uses a sequence of stages that transform a flow of data
- Some stages may retain state
- Pipeline connects tasks in a producer-consumer manner

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

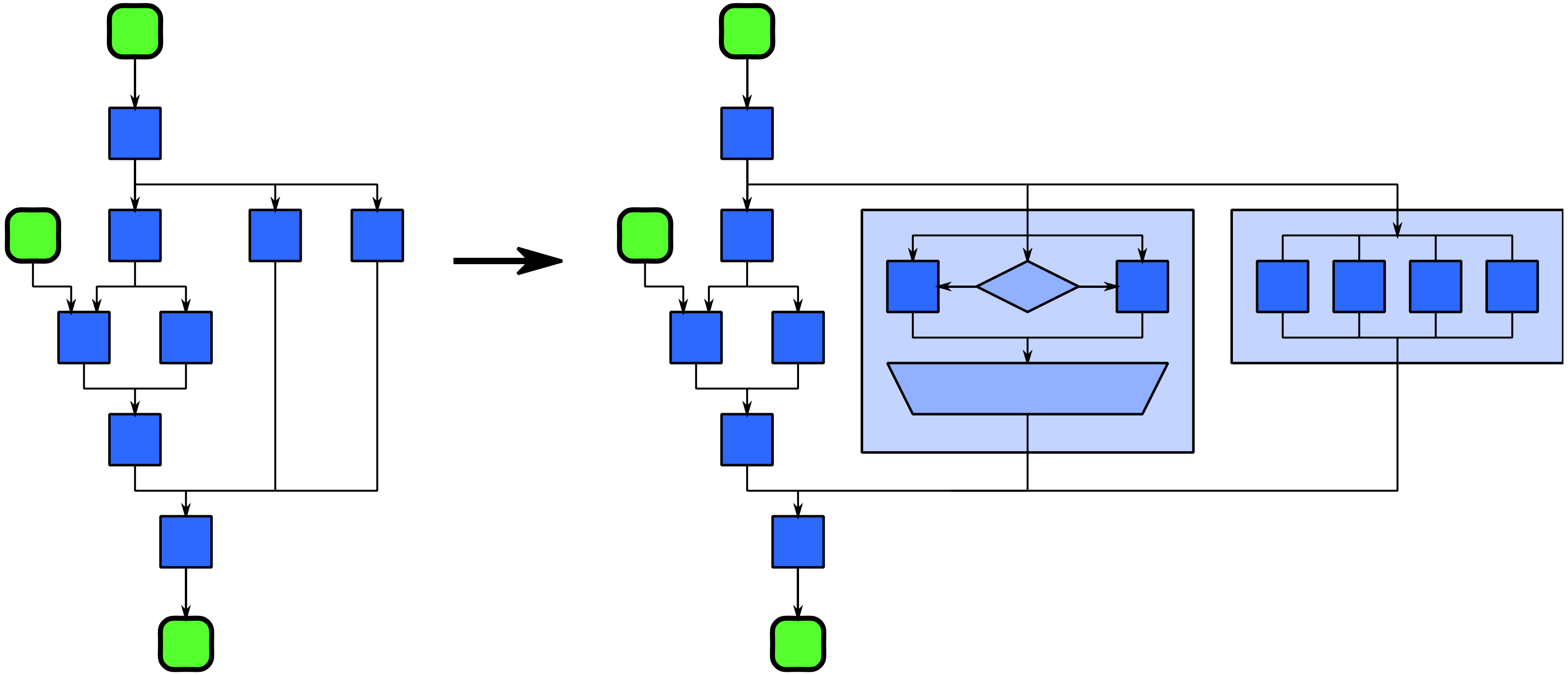
- Running multiple copies of stateless stages in parallel requires reordering of outputs

- Need to manage buffering between stages

Recursive Patterns

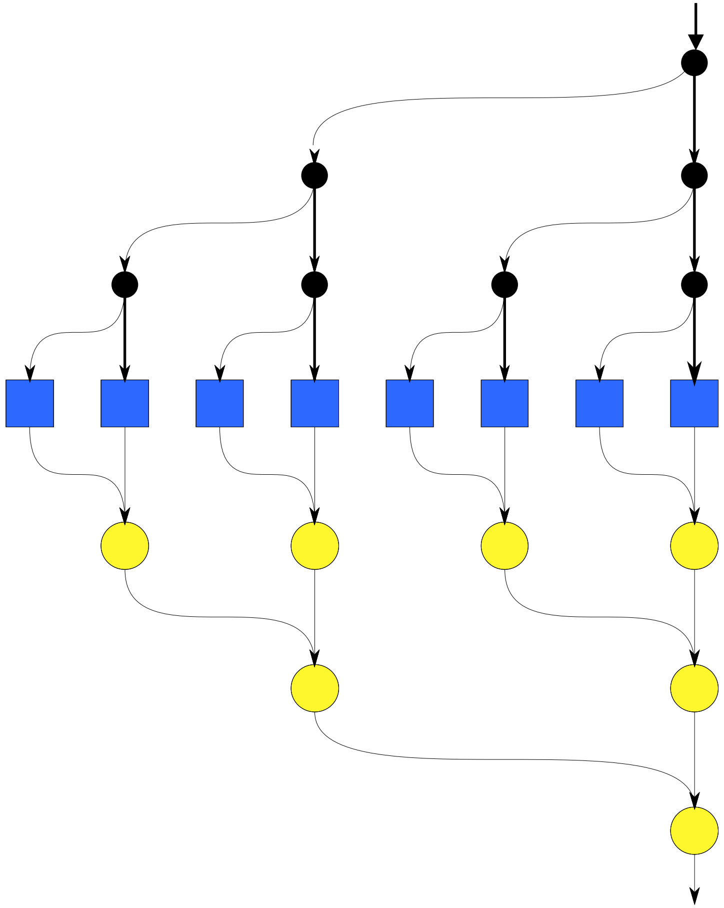
- Recursion is an important “universal” serial pattern
  - Recursion leads to functional programming
  - Iteration leads to procedural programming
- Structural recursion: nesting of components
- Dynamic recursion: nesting of behaviors

Nesting: Recursive Composition



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.

Fork-Join: Efficient Nesting



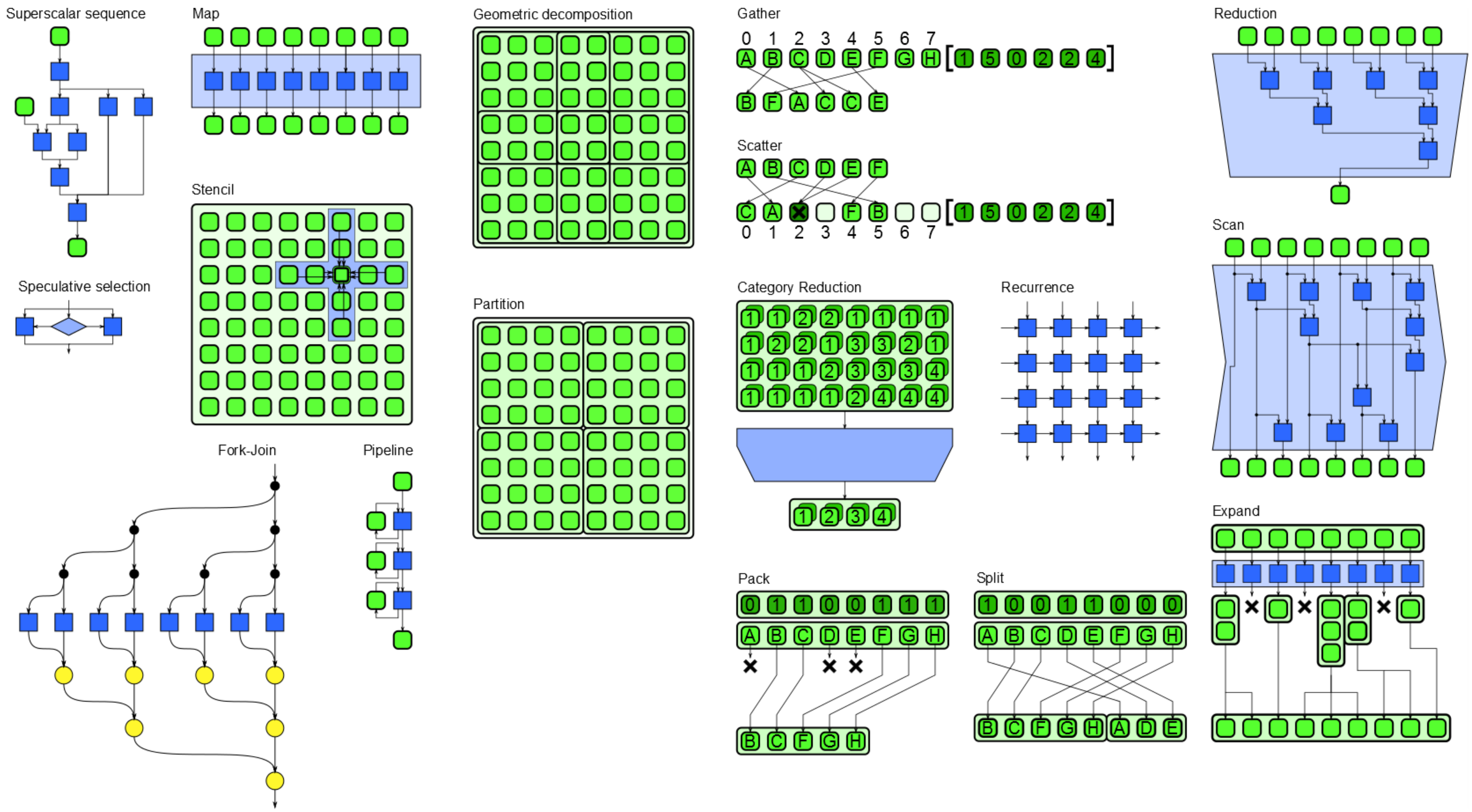
- Fork-join can be nested
- Spreads cost of work distribution and synchronization.

Recursive fork-join enables high parallelism.

Other Parallel Patterns

- **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
- **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

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*
- Parallelization may require reordering of operations
- 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

POSIX Threads

- POSIX standard multi-threading interface
  - For general multi-threaded concurrent programming
  - Defined as a set of **C** programming language types, functions and constants
  - Largely independent across implementations, and broadly supported
  - Common target for library and language implementation
- Provides primitives for
  - Thread management - creating, joining threads etc.
  - Synchronization
- POSIX Threads (pthreads) specification: http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html
- POSIX Threads Programming tutorial: https://computing.llnl.gov/tutorials/pthreads/

C++11 Multithreading

- C++11 standardizes support for multithreaded programming:
  - a memory model which allows multiple threads to co-exist in a program
  - library support for interaction between threads
- C++11 standard library includes:
  - Atomics
  - Threads
  - Mutexes
  - Conditional Variables
  - Futures and Promises
- C++11 multithreading reference: http://www.cplusplus.com/reference/multithreading/
- A good book on C++11 multithreading: **C++ Concurrency in Action: Practical Multithreading,** *by* Anthony Williams, Manning Publications, 2012

Grand Central Dispatch

- Developed by Apple
- Available on OS X 10.6 and later, iOS 4 and later
  - Open-sourced under the Apache license: https://libdispatch.macosforge.org
- An implementation of **task parallelism** based on the **thread pool** pattern
  - Still uses threads at the low level but abstracts them away from the programmer
  - Allows tasks to be queued, then schedules them to execute on any of the available processor cores
  - A task can be expressed either as a function or as a “block”
  - Grand Central Dispatch (GCD) reference: https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/

Thread Building Blocks

- Threading Building Blocks (TBB) is a **C++** template library developed by Intel for parallel programming on multi-core processors
  - A TBB program specifies graphs of dependent tasks according to *algorithms/patterns*, instead of manipulating threads
  - TBB implements *work stealing* to balance a parallel workload across available processing cores in order to increase core utilization and therefore scaling
  - TBB includes efficient low-level primitives (atomics, memory allocation, etc.)
- TBB is available:
  - both commercially as a binary distribution with support
  - and as open-source software: https://www.threadingbuildingblocks.org/
- TBB tutorial: https://www.threadingbuildingblocks.org/intel-tbb-tutorial

TBB 4.0 Components

Parallel Algorithms
parallel_for
parallel_for_each
parallel_invoke
parallel_do
parallel_scan
parallel_sort
parallel_[deterministic]_reduce

Macro Dataflow
parallel_pipeline
tbb::flow::...

Task scheduler
task_group, structured_task_group
task
task_scheduler_init
task_scheduler_observer

Synchronization Primitives
atomic, condition_variable
[recursive_]mutex
{spin,queuing,null} [_rw]_mutex
critical_section, reader_writer_lock

Threads
std::thread

Concurrent Containers
concurrent_hash_map
concurrent_unordered_{map,set}
concurrent_[bounded_]queue
concurrent_priority_queue
concurrent_vector

Thread Local Storage
combinable
enumerable_thread_specific

Memory Allocation
tbb_allocator
zero_allocator
cache_aligned_allocator
scalable_allocator

Intel Cilk Plus

- Intel Cilk Plus is an *extension* to **C** and **C++** that simplifies the expression of task and data parallelism:
  - Fork-join parallel programming model
  - Serial semantics if keywords are ignored (serial elision)
  - Efficient work-stealing load balancing
  - Supports vector parallelism via array slices and elemental functions
- Cilk Plus is available:
  - both commercially as a binary distribution with support
  - and as open-source software: https://www.cilkplus.org/
- Cilk Plus tutorial: https://www.cilkplus.org/cilk-plus-tutorial

Summary of Intel Cilk Plus

Thread Parallelism
cilk_spawn
cilk_sync
cilk_for

Vector Parallelism
array notation
#pragma simd
elemental functions

Reducers
reducer
reducer_op{add,and,or,xor}
reducer_{min,max}{_index}
reducer_list_{append,prepend}
reducer_ostream
reducer_string
holder