departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

# Parallel Programming Patterns Overview

Concurrency and Parallelism — 2019-20

Master in Computer Science

(Mestrado Integrado em Eng. Informática)

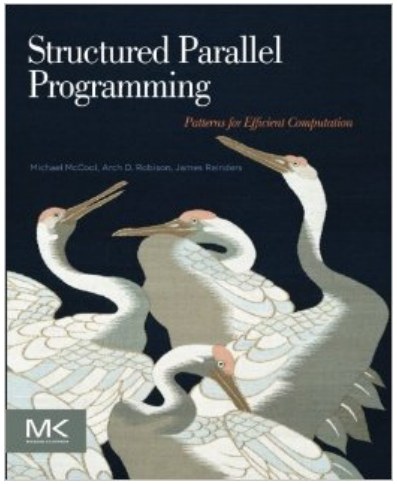
Joao Lourenço <joao.lourenco@fct.unl.pt>

Source: Parallel Computing, CIS 410/510, Department of Computer and Information Science

# Outline

- Structured programming patterns overview
  - Concept of programming patterns
  - Serial and parallel control flow patterns
  - Serial and parallel data management patterns

  - Bibliography:
    - **Chapter 3** of book
      McCool M., Arch M., Reinders J.;
      Structured Parallel Programming: Patterns for Efficient Computation;
      Morgan Kaufmann (2012);
      ISBN: 978-0-12-415993-8



# 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

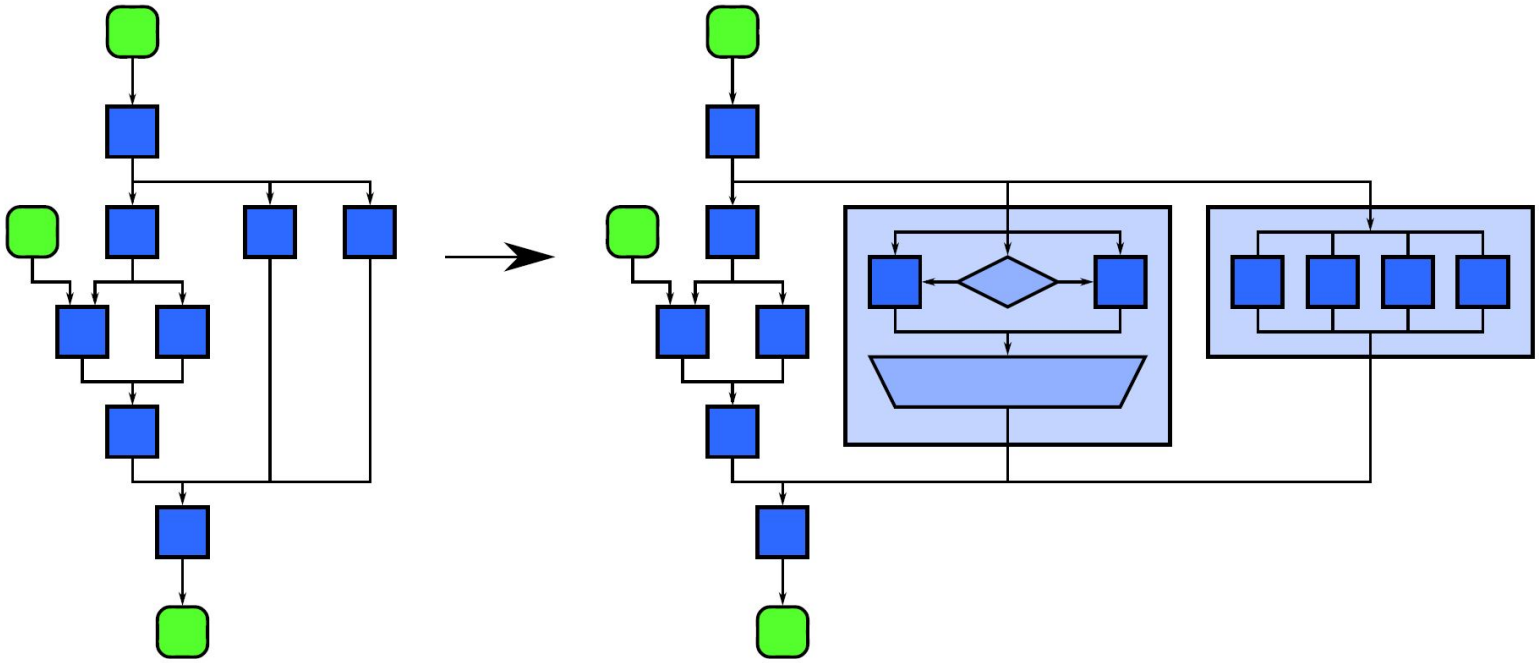
# Parallel Patterns

- Nesting Pattern
- Serial / Parallel Control Patterns
- Serial / Parallel Data Management Patterns
- Other Patterns
- Programming Model Support for Patterns

# Nesting Pattern

- **Nesting** is the ability to hierarchically compose patterns
- This pattern appears in both serial and parallel algorithms
- "Pattern diagrams" are used to visually show the pattern idea where each "task block" is a location of general code in an algorithm
- Each "task block" can in turn be another pattern in the **nesting pattern**

# 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.

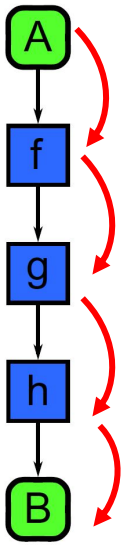
# Serial Control Patterns

- Structured serial programming is based on these patterns: **sequence**, **selection**, **iteration**, and **recursion**
- The **nesting** pattern can also be used to hierarchically compose these four patterns
- Though you should be familiar with these, it's extra important to understand these patterns when parallelizing serial algorithms based on these patterns

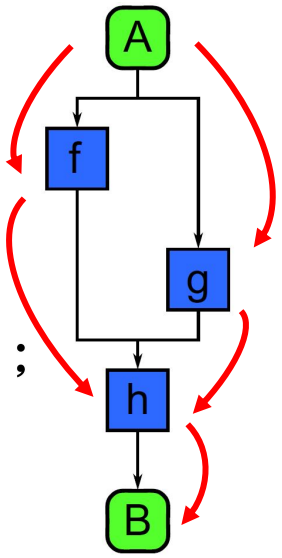
# Serial Control Patterns: Sequence

- **Sequence**: Ordered list of tasks that are executed in a specific order
- Assumption – program text ordering will be followed (seems obvious… but this will be important when parallelized)

```
1  T = f(A);
2  S = g(T);
3  B = h(S);
```

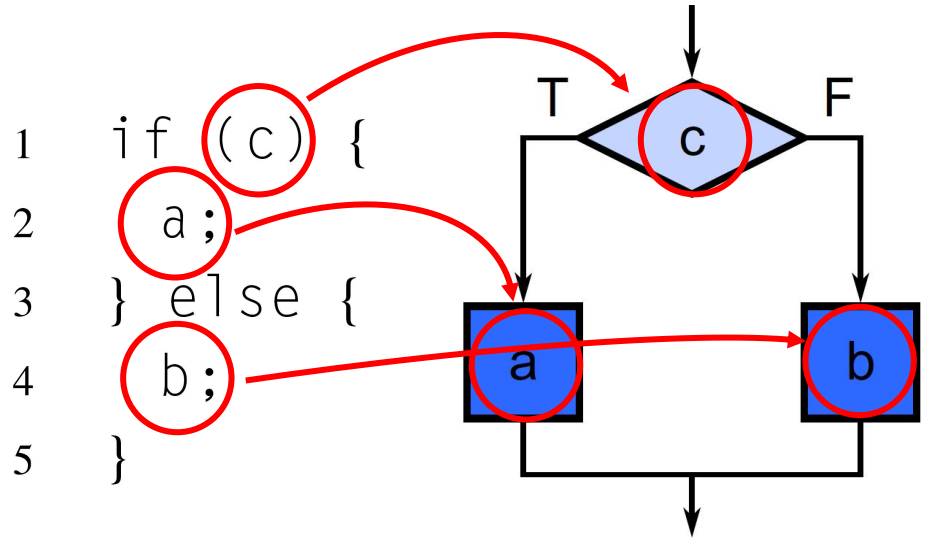



```
1  T = f(A);
2  S = g(A);
3  B = h(S,T);
```



# Serial Control Patterns: Selection

- **Selection**: condition *c* is first evaluated. Either task *a* or *b* is executed depending on the true or false result of *c*.
- Assumptions – *a* and *b* are never executed before *c*, and either *a* or *b* is executed — never both

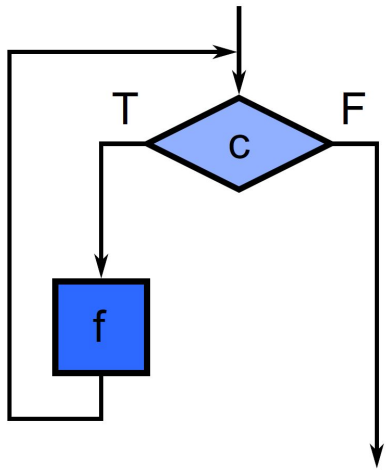


# Serial Control Patterns: Iteration

- **Iteration**: condition *c* is evaluated. If it is true, *a* is evaluated, and then *c* is evaluated again. This repeats until *c* is false.
- Complication when parallelizing: potential for dependencies to exist between previous iterations

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

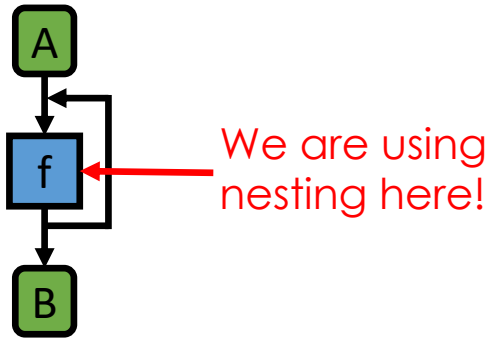
```
while (c) {
  f();
}
```



# Serial Control Patterns: Recursion

- **Recursion**: dynamic form of nesting allowing functions to call themselves
- Tail recursion is a special recursion that can be converted into iteration – important for functional languages

```
int fact (int n) {
  if (n == 0)
    return 1
  else
    return n*fact (n-1);
}
```

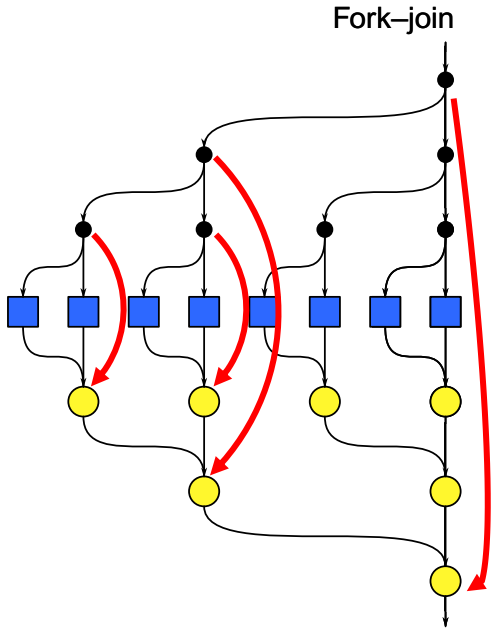


# Parallel Control Patterns

- 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



# Parallel Control Patterns: Fork-Join

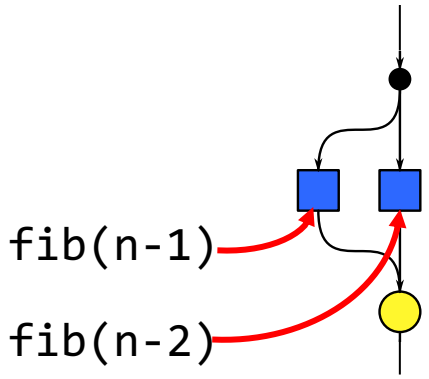
- **Fork-join**: allows control flow to fork into multiple parallel flows, then rejoin later

**Parallel — Cilk+**

```
int fib(int n)
{
    if (n < 2)
        return n;
    int x = cilk_spawn fib(n-1);
    int y = fib(n-2);
    cilk_sync;
    return x + y;
}
```

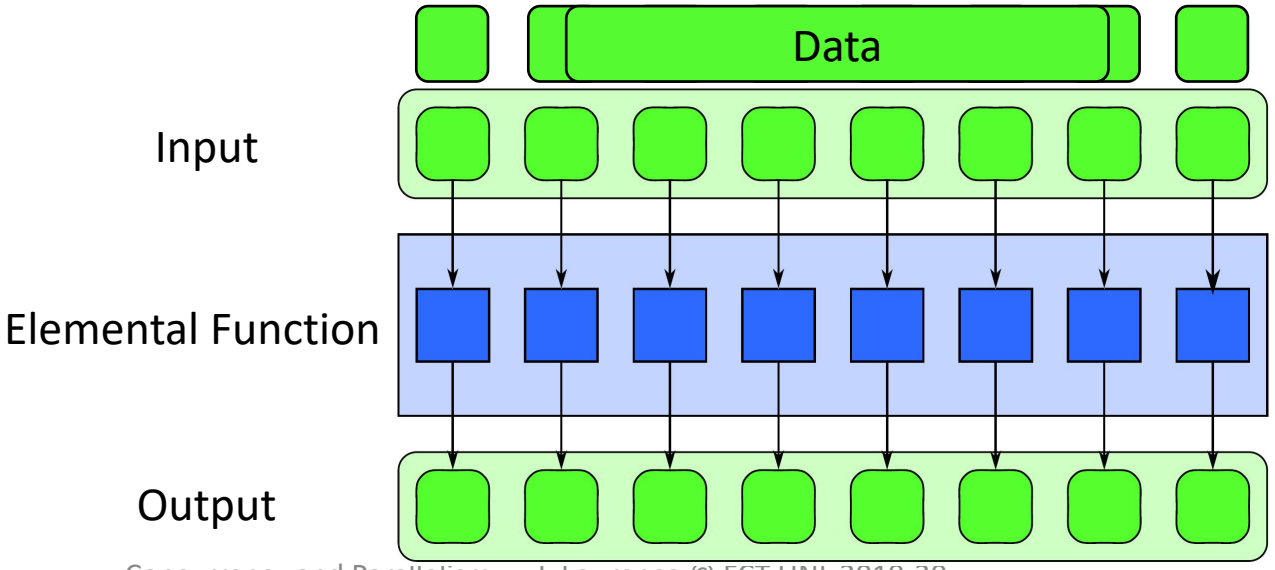
**Sequential**

```
int fib(int n)
{
    if (n < 2)
        return n;
    int x = fib(n-1);
    int y = fib(n-2);
    return x + y;
}
```



# 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: Map

- **Map**: performs a function over every element of a collection

| Sequential | Cilk+ |
| --- | --- |
| | |
| | |
| | |
| | |
| | |
| | |

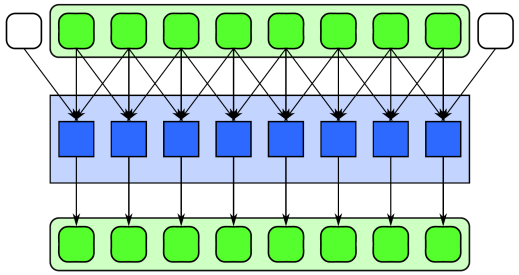
# Parallel Control Patterns: Map

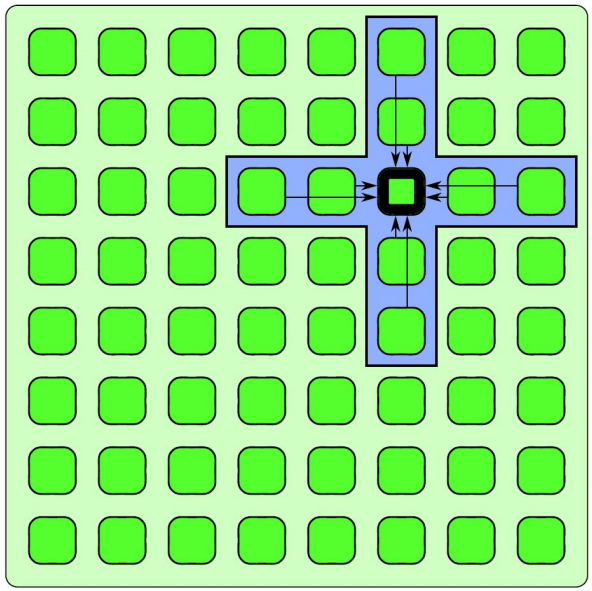
- **Map**: performs a function over every element of a collection

| Sequential | Cilk+ |
|---|---|
| `for (i = 0; i < 10; i++)`<br>`  A[i] = 5;` | `A[:] = 5;` |
| `for (i = 0; i < 10; i++)`<br>`  A[i] = B[i];` | `A[:] = B[:];` |
| `for (i = 0; i < 10; i++)`<br>`  A[i] = B[i] + 1;` | `A[:] = B[:] + 1;` |
| `for (i = 0; i < 10; i++)`<br>`  D[i] = A[i] + B[i];` | `D[:] = A[:] + B[:];` |
| `for (j = 0; j < 10; j++)`<br>`  C[X][j] = A[j]` | `C[X][:] = A[:];` |
| `for (i = 0; i < 10; i++)`<br>`  func (A[i]);` | `func (A[:]);` |

# Parallel Control Patterns: Stencil

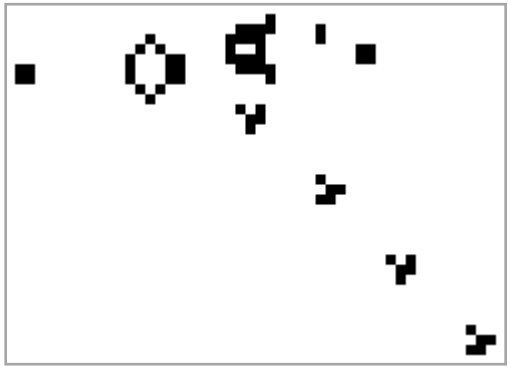
- **Stencil**: Elemental function accesses a set of "neighbors", stencil is a generalization of map
- Often combined with iteration – used with iterative solvers or to evolve a system through time
- Boundary conditions must be handled carefully in the stencil pattern





# Conway's Game of Life

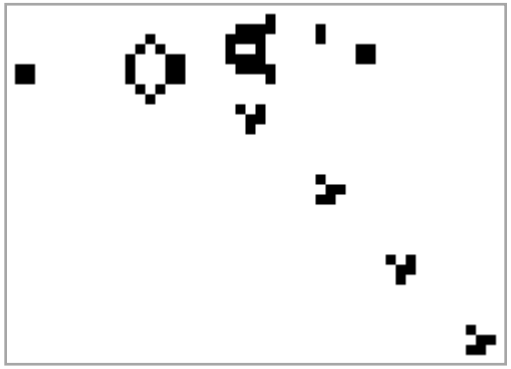
- The **Game of Life** is a cellular automaton created by John Conway in 1970
- The evolution of the game is entirely based on the input state – zero player game
- To play: create initial state, observe how the system evolves over successive time steps



2D landscape

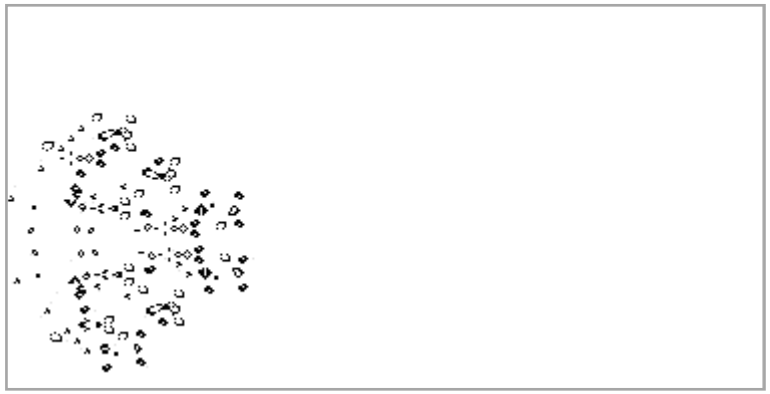
# Conway's Game of Life

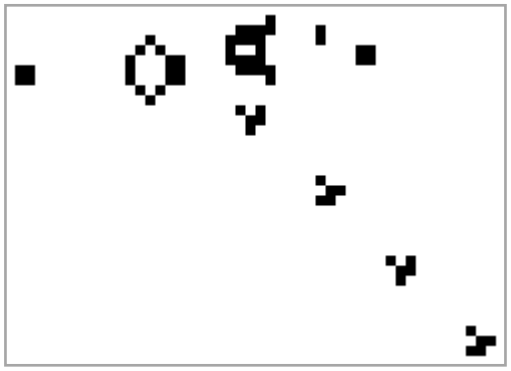
- Typical rules for the Game of Life
  - Infinite 2D grid of square cells, each cell is either "alive" or "dead"
  - Each cell will interact with all 8 of its neighbors
    - Any cell with < 2 live neighbors dies (under-population)
    - Any cell with 2 or 3 live neighbors lives to next gen.
    - Any cell with > 3 live neighbors dies (overcrowding)
    - Any dead cell with 3 live neighbors becomes a live cell

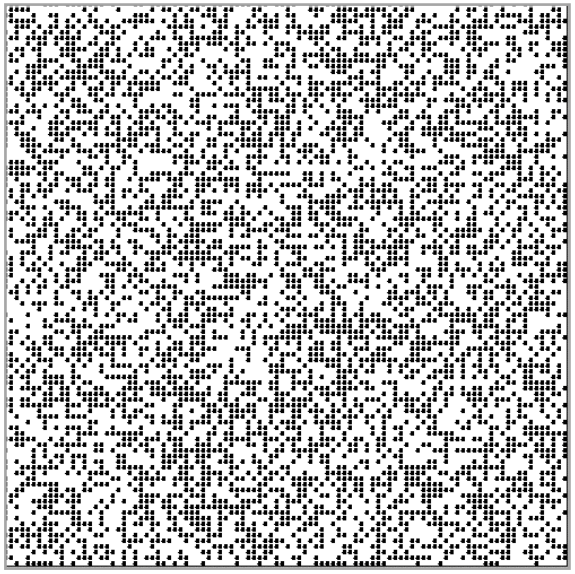


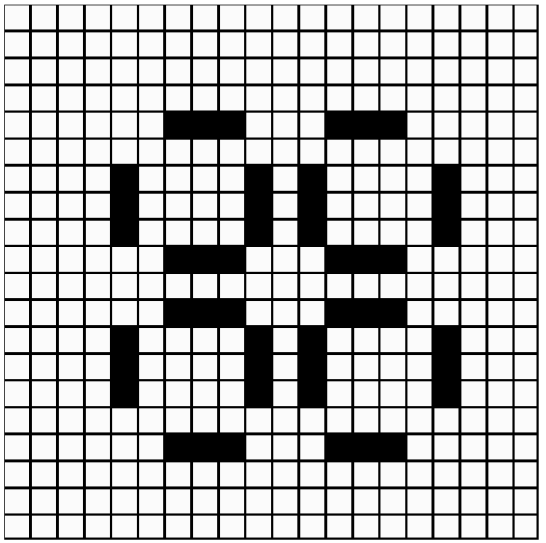
2D landscape

# Conway's Game of Life: Examples



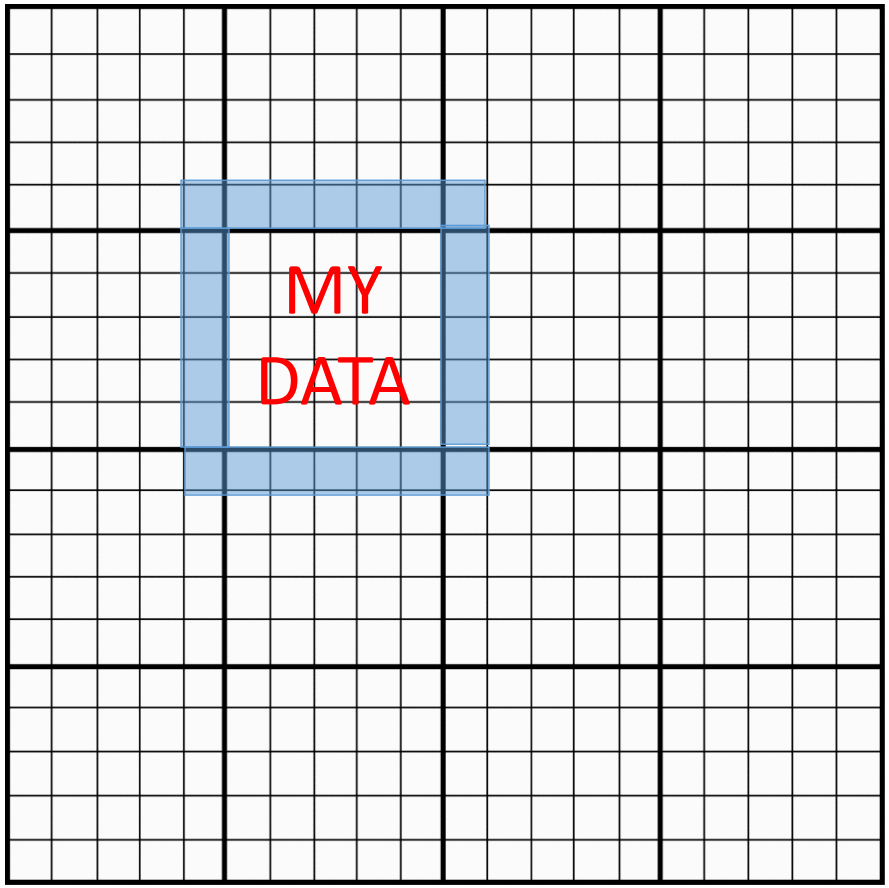






# Conway's Game of Life

- The Game of Life computation can easily fit into the stencil pattern!
- Each larger, black box is owned by a thread
- What will happen at the boundaries?

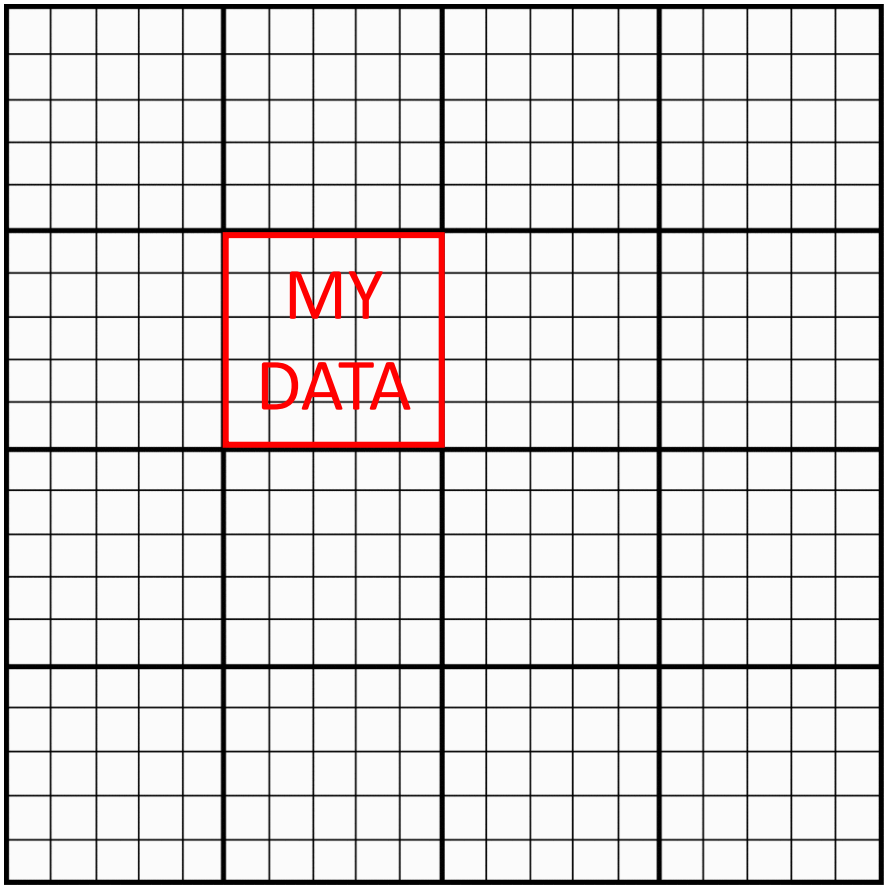


# Conway's Game of Life

- We need some way to preserve information from the previous iteration without overwriting it
- **Ghost Cells** are one solution to the boundary and update issues of a stencil computation
- Each thread keeps a copy of neighbors' data to use in its local computations
- These ghost cells must be updated after each iteration of the stencil

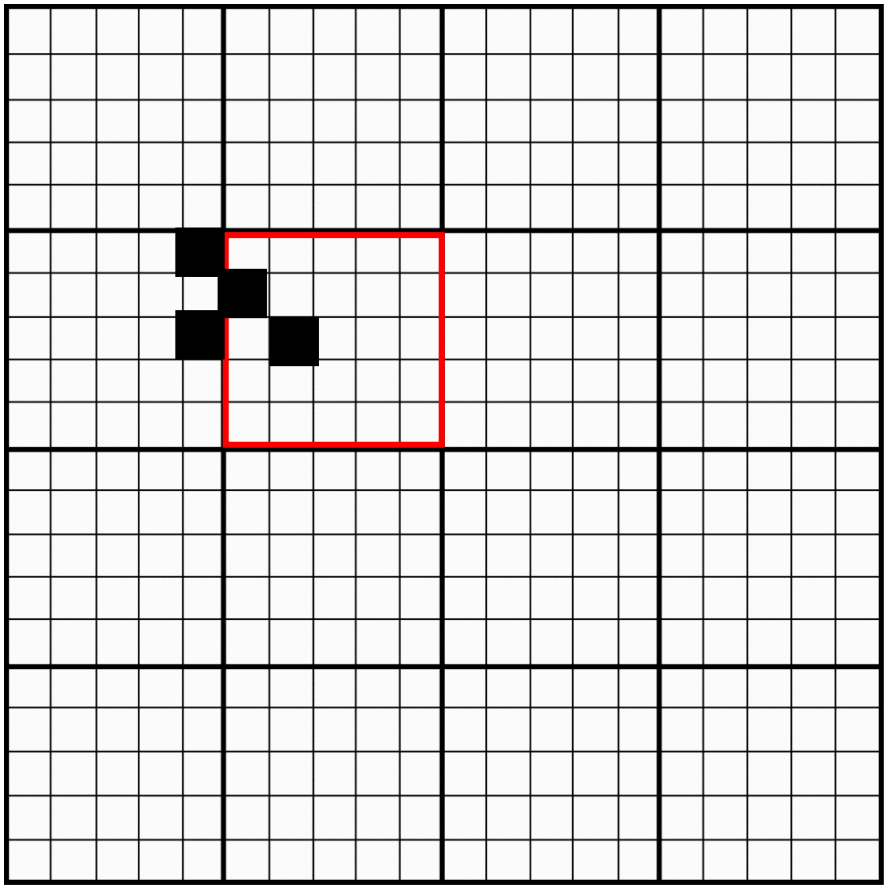
# Conway's Game of Life

- Working with ghost cells



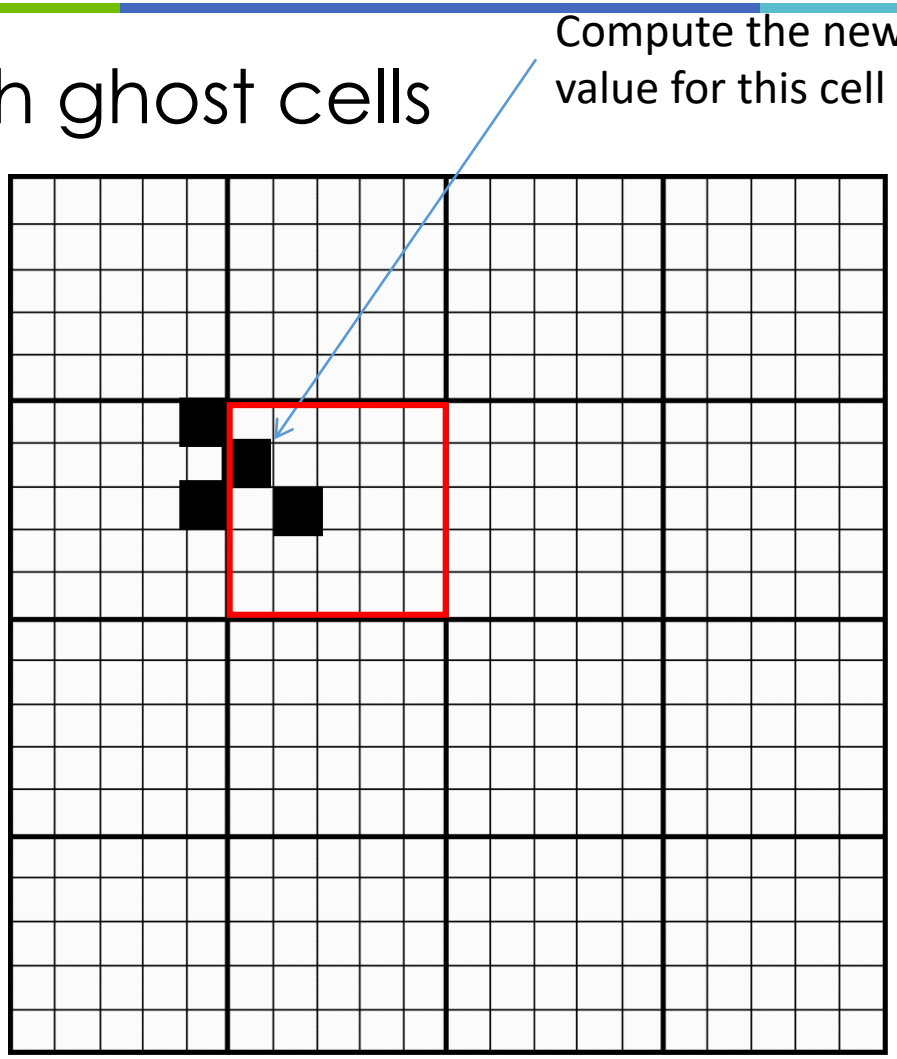
# Conway's Game of Life

- Working with ghost cells



# Conway's Game of Life

- Working with ghost cells

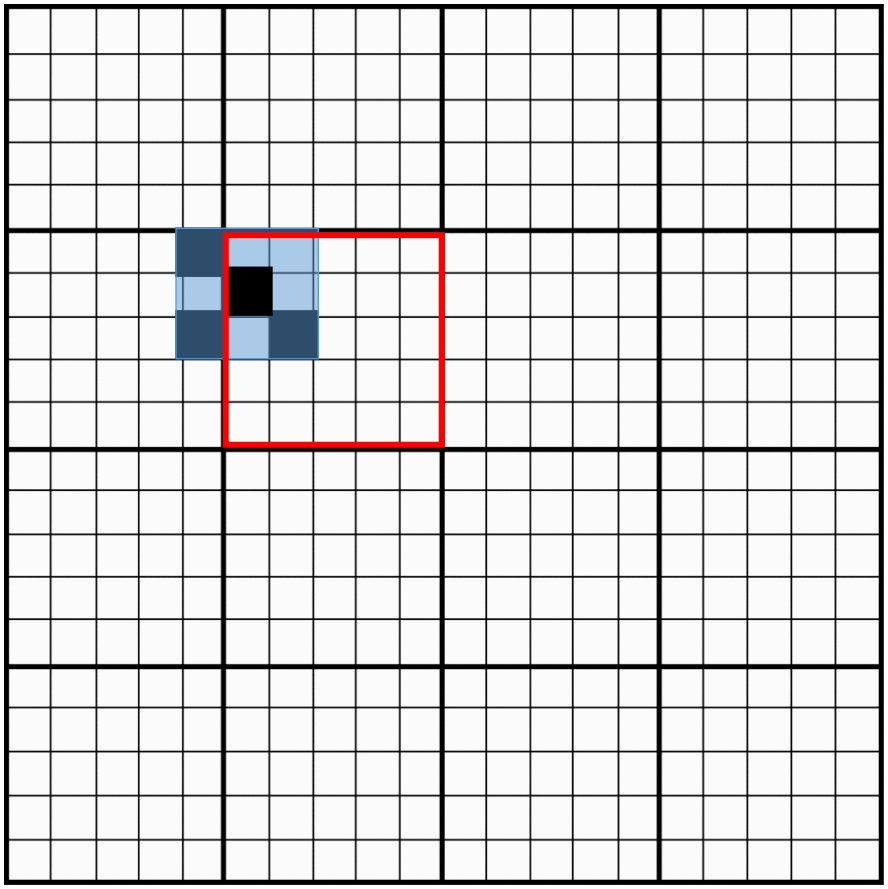


# Conway's Game of Life

- Working with ghost cells

Five of its eight neighbors already belong to this thread

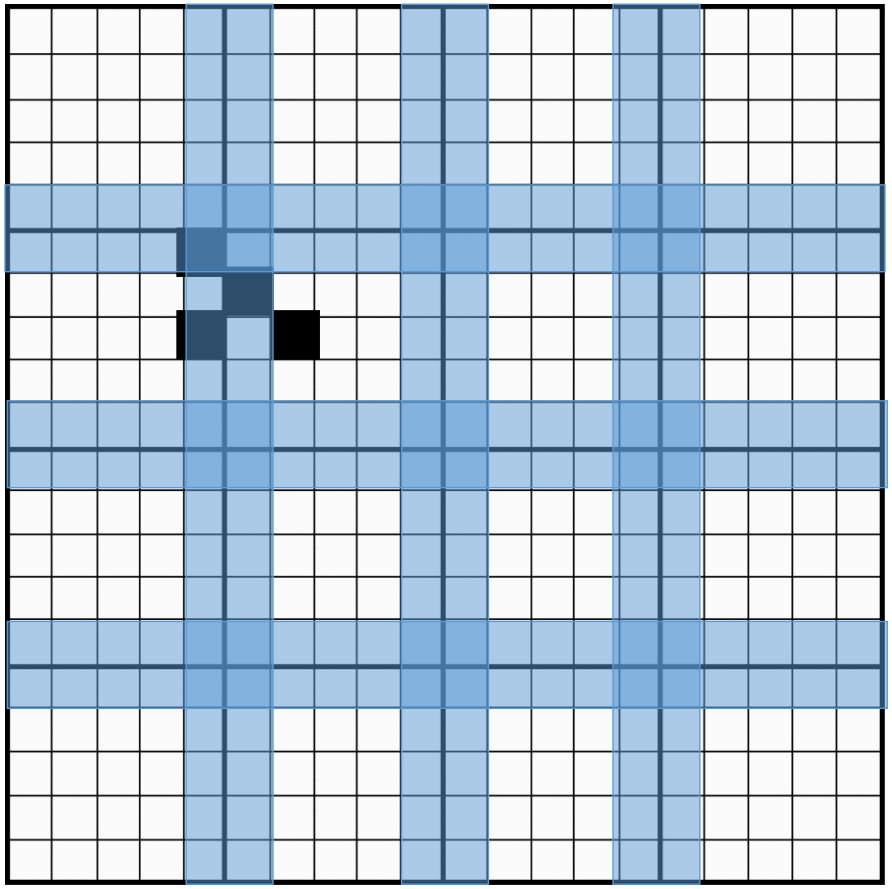
But three of its neighbors belong to a different thread



# Conway's Game of Life

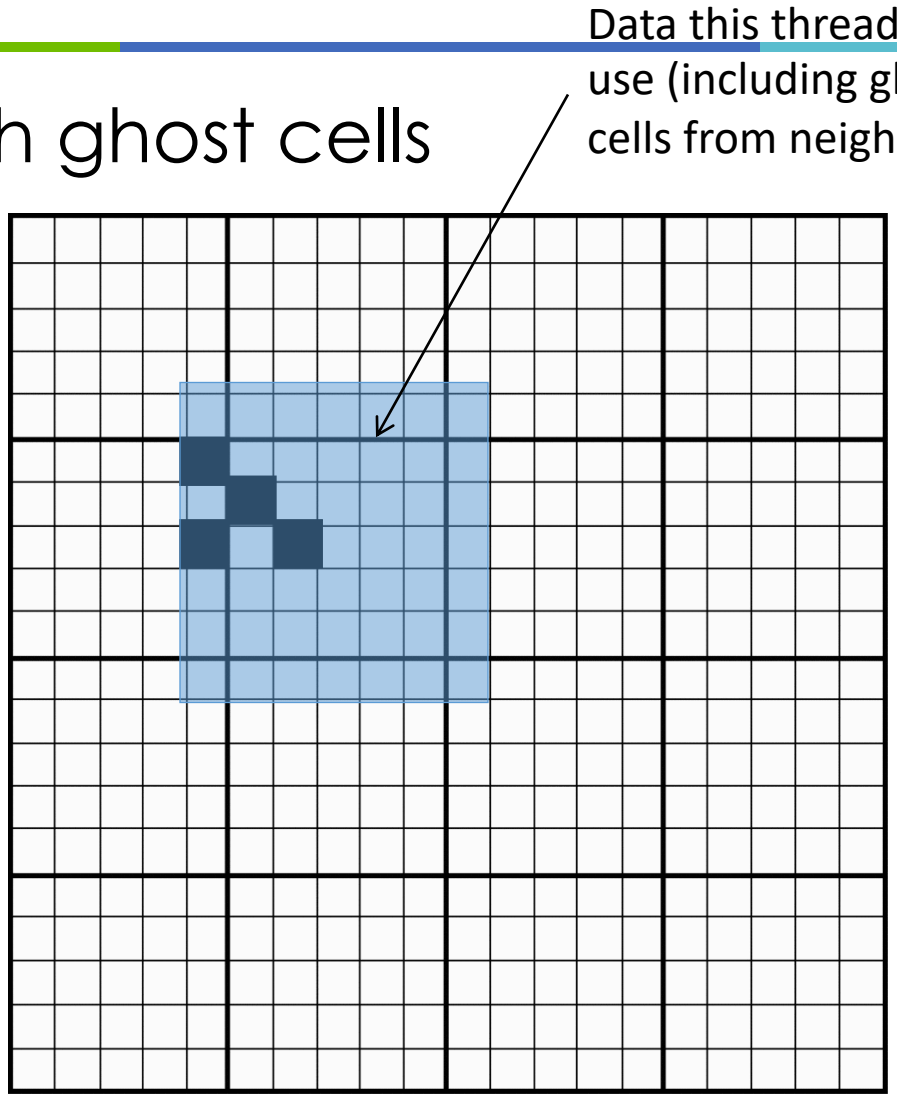
- Working with ghost cells

Before any updates are done in a new iteration, all threads must update their ghost cells



# Conway's Game of Life

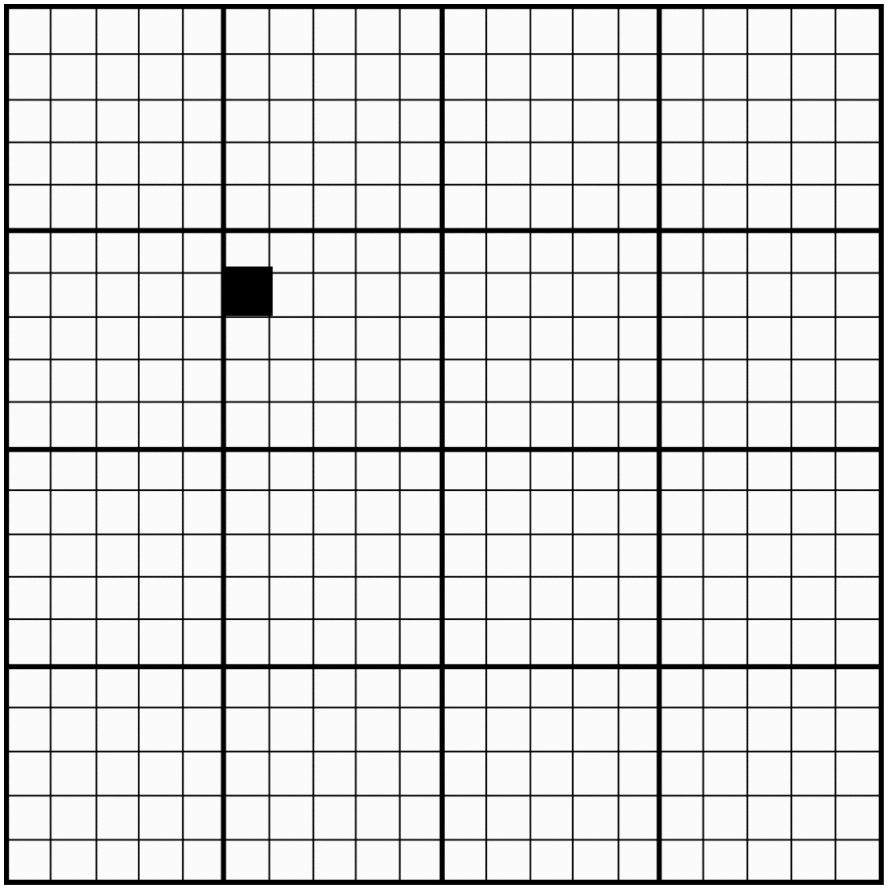
- Working with ghost cells



# Conway's Game of Life

- Working with ghost cells

Updated cells



# Conway's Game of Life

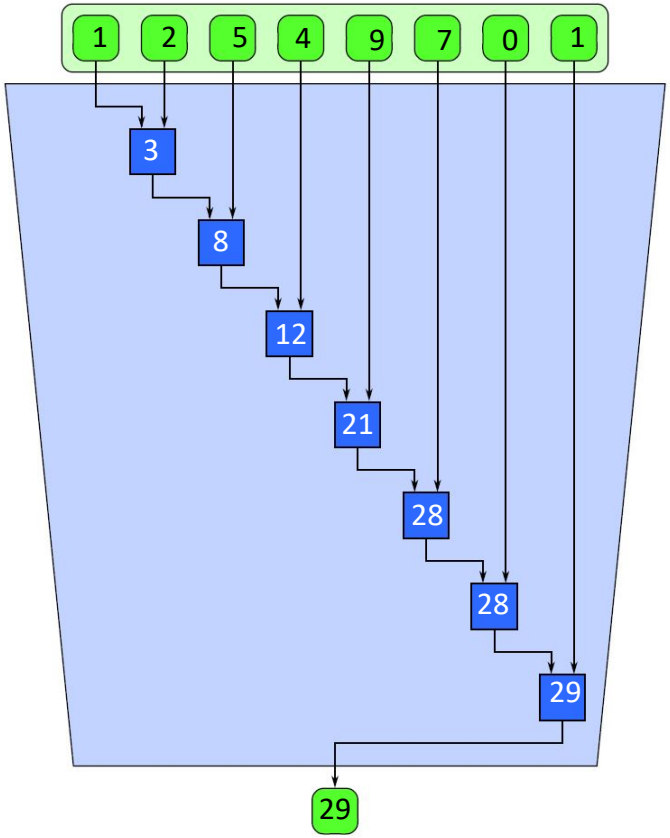
- Things to consider…
  - What might happen to our ghost cells as we increase the number of threads?
    - the ghost cells to total cells ratio will rapidly increase causing
      - a greater demand on memory
      - larger overhead (ratio management/computation)
  - What would be the benefits of using a larger number of ghost cells per thread? Negatives?
    - in the Game of Life example, we could double or triple our ghost cell boundary, allowing us to perform several iterations without stopping for a ghost cell update

# Parallel Control Patterns: Reduction

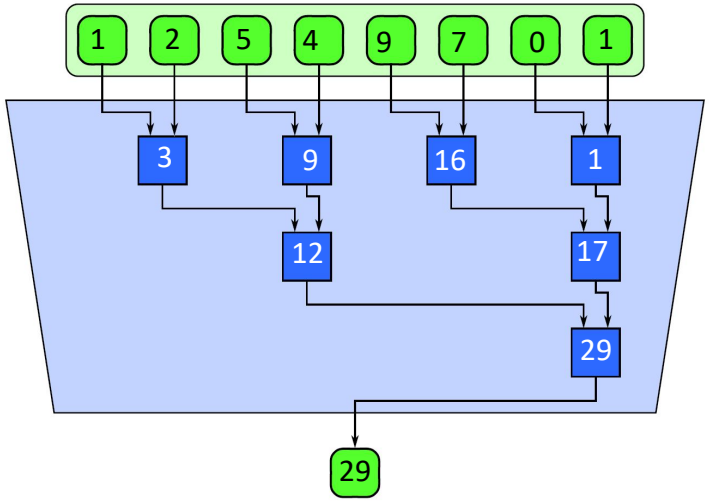
- **Reduction**: Combines every element in a collection using an associative "combiner function"
- Because of the associativity of the combiner function, different orderings of the reduction are possible
- Examples of combiner functions: addition, multiplication, maximum, minimum, and Boolean AND, OR,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

# Parallel Control Patterns: Reduction

Serial Reduction




Parallel Reduction

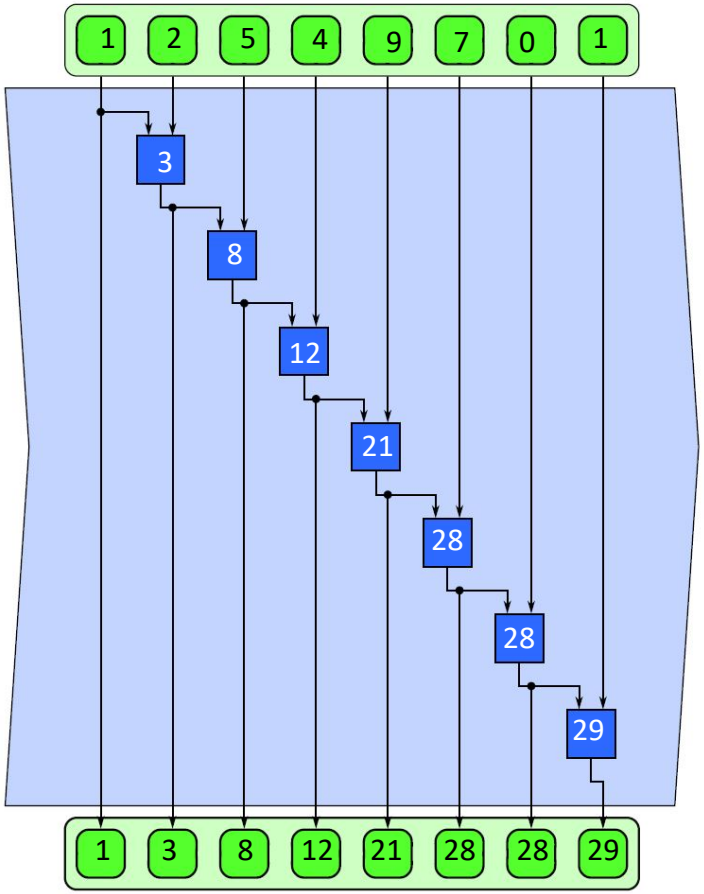


# Parallel Control Patterns: Scan

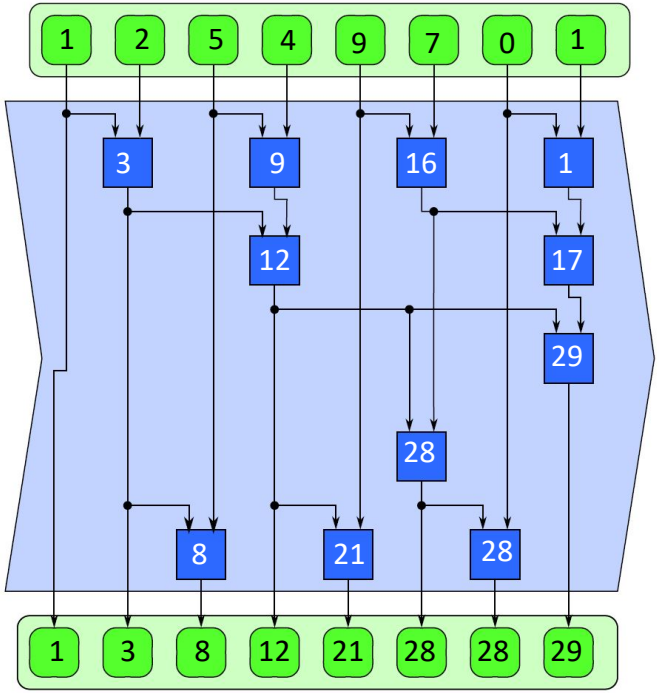
- **Scan**: computes all partial reductions of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- If the function being used is associative, the scan can be parallelized
- Parallelizing a scan is not obvious at first, because of dependencies to previous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

# Parallel Control Patterns: Scan

Serial Scan

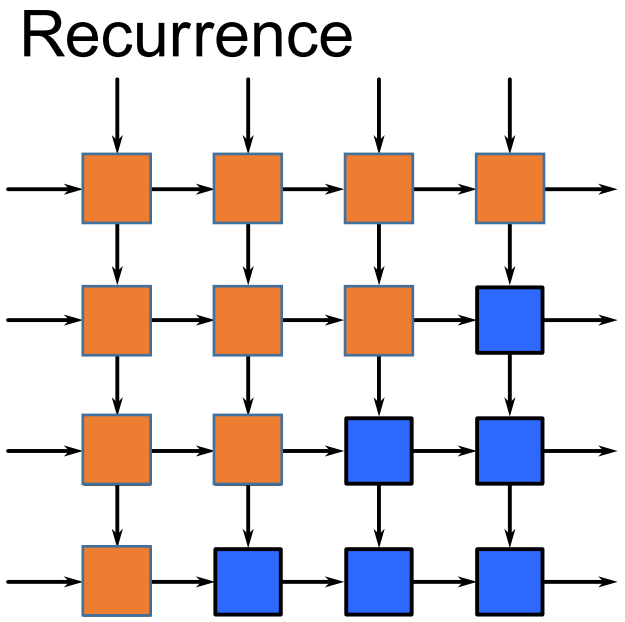



Parallel Scan



# 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

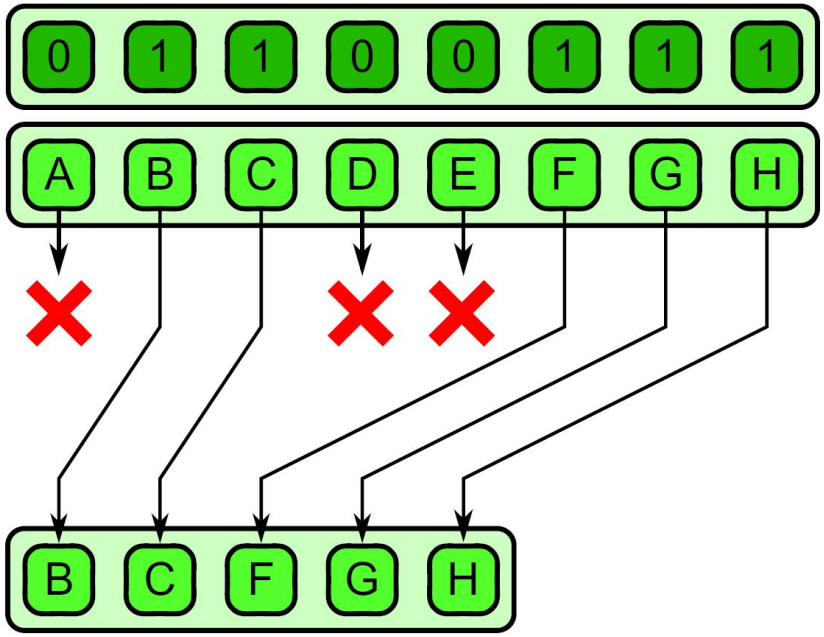
- Objects are language constructs to associate data with code to manipulate and manage that data
- Objects can have member functions, and they also are considered members of a class of objects
- Parallel programming models will generalize objects in various ways

# 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**

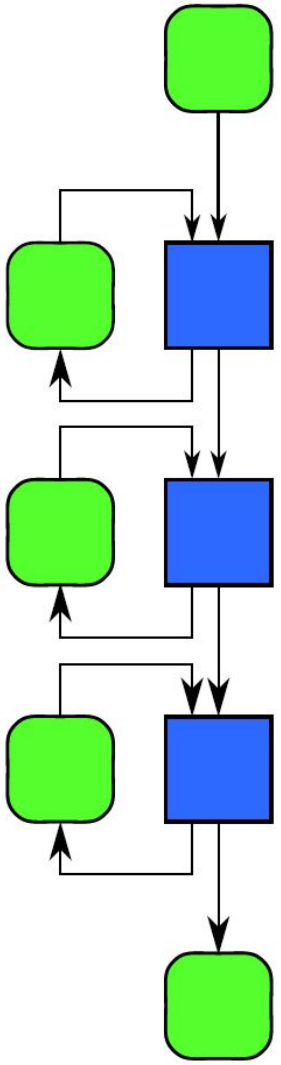
# Parallel Data Management Patterns: Pack

- **Pack** is used to eliminate unused space in a collection (like a filter)
- 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



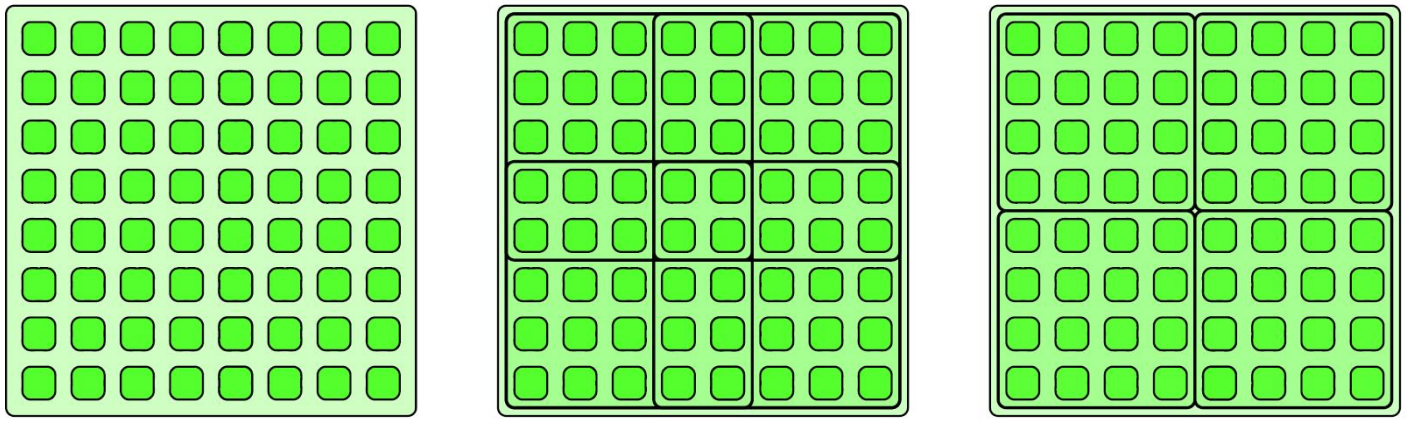
# Parallel Data Management Patterns: Pipeline

- **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



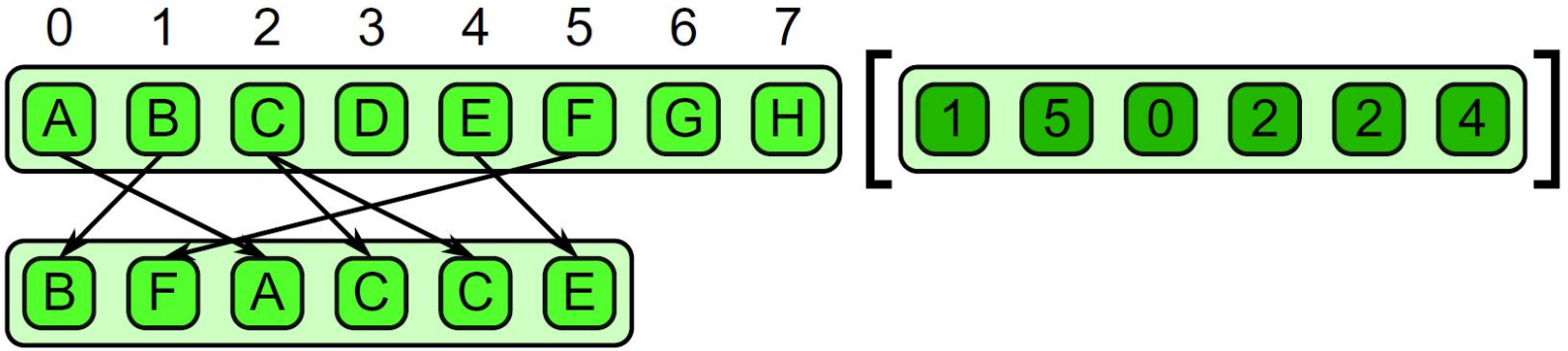
# Parallel Data Management Patterns: Geometric Decomposition

- **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



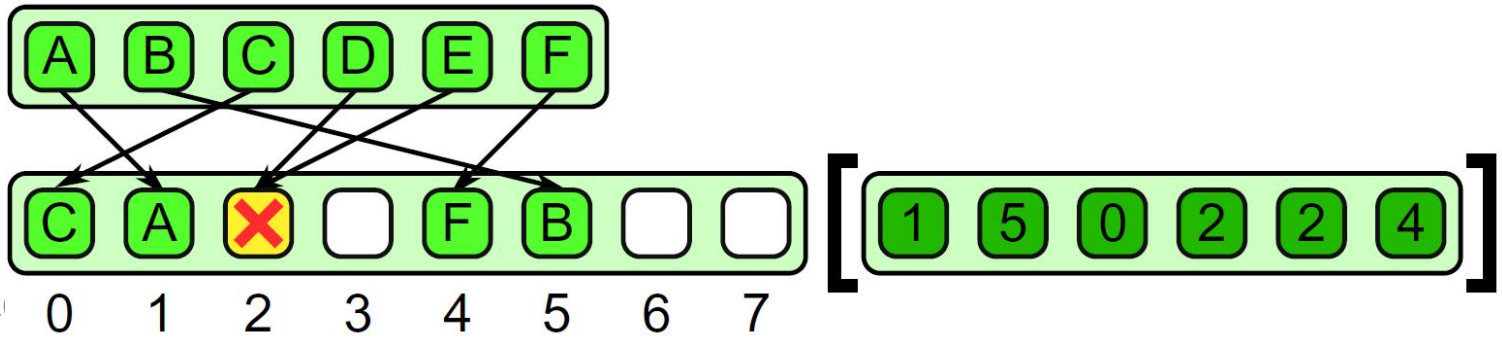
# Parallel Data Management Patterns: Gather

- **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



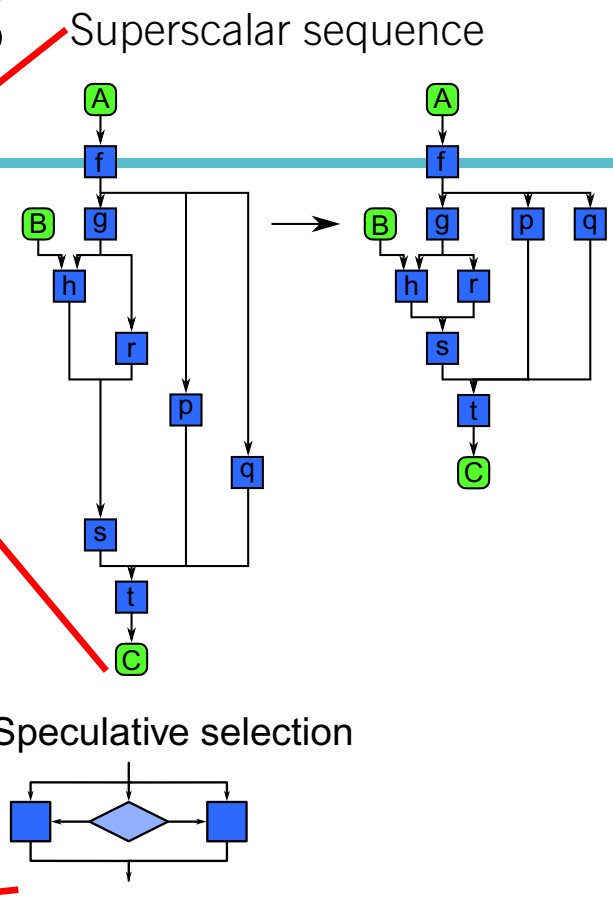
# Parallel Data Management Patterns: Scatter

- **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

# Overview of Parallel Patterns

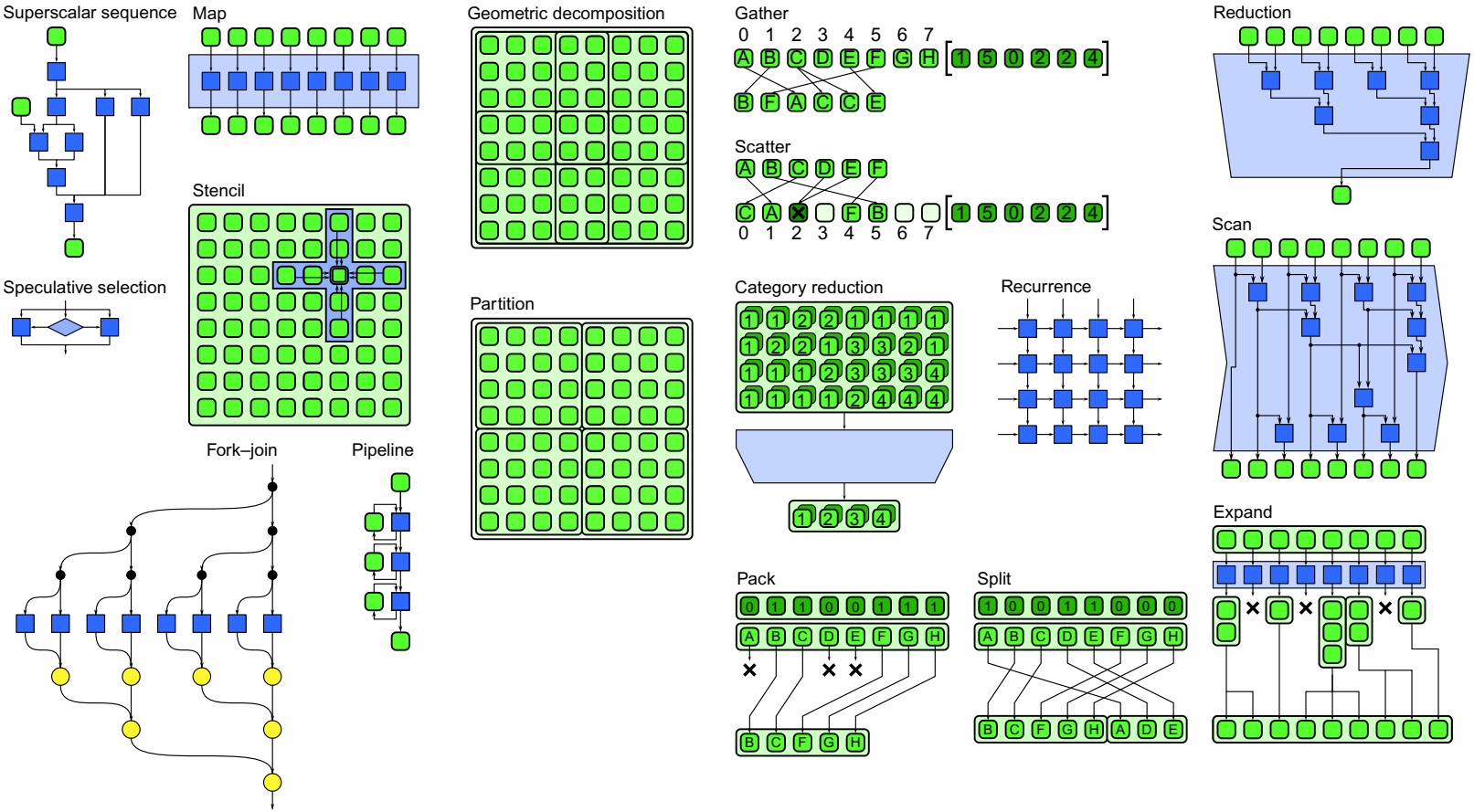



**FIGURE 1.11**

Overview of parallel patterns.

# The END