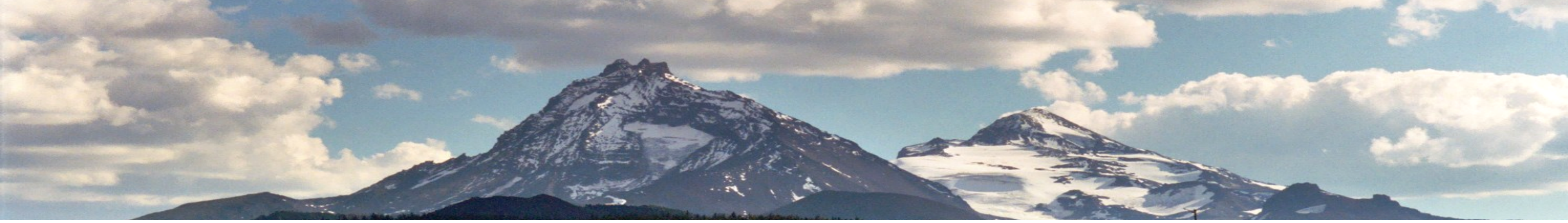


Parallel Programming Patterns Overview and Map Pattern

Parallel Computing

CIS 410/510

Department of Computer and Information Science

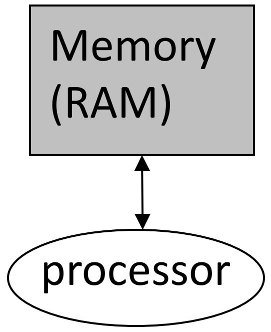
UNIVERSITY OF OREGON

Outline

- Parallel programming models
- Dependencies
- Structured programming patterns overview
  - Serial / parallel control flow patterns
  - Serial / parallel data management patterns
- Map pattern
  - Optimizations
    - sequences of Maps
    - code Fusion
    - cache Fusion
  - Related Patterns
  - Example: Scaled Vector Addition (SAXPY)

Parallel Models 101

- Sequential models
  - von Neumann (RAM) model

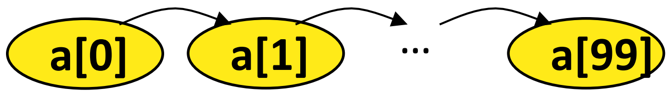


- Parallel model
  - A parallel computer is simple a collection of *processors interconnected* in some manner to *coordinate* activities and *exchange data*
  - Models that can be used as general frameworks for describing and analyzing parallel algorithms
    - *Simplicity*: description, analysis, architecture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

Directed Acyclic Graphs (DAG)

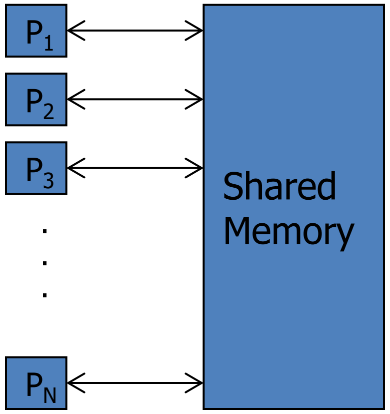
- Captures data flow parallelism
- Nodes represent operations to be performed
  - Inputs are nodes with no incoming arcs
  - Output are nodes with no outgoing arcs
  - Think of nodes as tasks
- Arcs are paths for flow of data results
- DAG represents the operations of the algorithm and implies precedent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

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



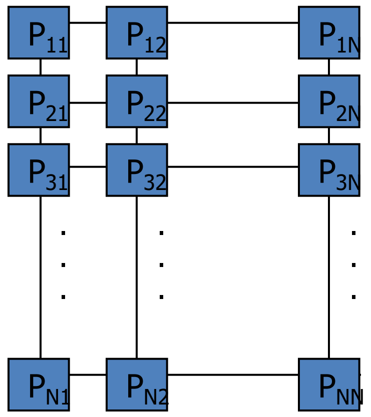
Shared Memory Model

- Parallel extension of RAM model (PRAM)
  - Memory size is infinite
  - Number of processors in 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 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 different machines
- 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-grain versus fine-grainded
- Thinking about parallelism
- Evaluation

Why is parallel programming important?

- Parallel programming has matured
  - Standard programming models
  - Common machine architectures
  - Programmer can focus on computation and use suitable programming model for implementation
- Increase portability between models and architectures
- Reasonable hope of portability across platforms
- Problem
  - Performance optimization is still platform-dependent
  - Performance portability is a problem
  - Parallel programming methods are still evolving

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
- Which are hard to do?

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:

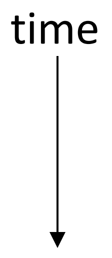
Processor 1:	Processor 2:
Statement 1;	Statement 2;

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

Sequential Consistency in Parallel Execution

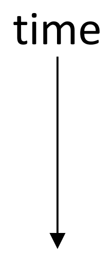
- Case 1:

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



- Case 2:

```
Processor 1:        Processor 2:
                      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,

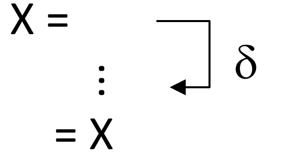
  S1;

  S2;

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

  if and only if

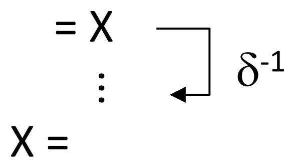
  S2 reads a value written by S1



- S2 has a *anti-dependence* on S1

  if and only if

  S2 writes a value read by S1



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

$$\begin{tikzcd} X = \arrow[dd, bend left, "\delta^0"] \\ \vdots \\ X = \end{tikzcd}$$

- 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

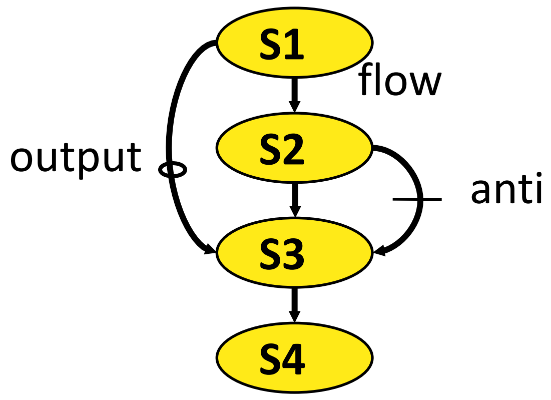
- Can use graphs to show dependence relationships
- Example

  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

$out(S_1) \cap in(S_2) \neq \emptyset$	$S_1 \, \delta \; S_2$	flow dependence
$in(S_1) \cap out(S_2) \neq \emptyset$	$S_1 \, \delta^{-1} \, S_2$	anti-dependence
$out(S_1) \cap out(S_2) \neq \emptyset$	$S_1 \, \delta^{0} S_2$	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;
}
```

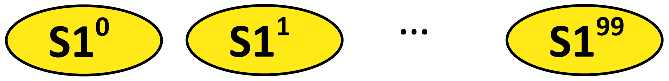
- Dependencies? What about *i*, the loop index?
- *DOALL* loop (a.k.a. *foreach* loop)
  - All iterations are independent of each other
  - All statements be executed in parallel at the same time
    - Is this really true?

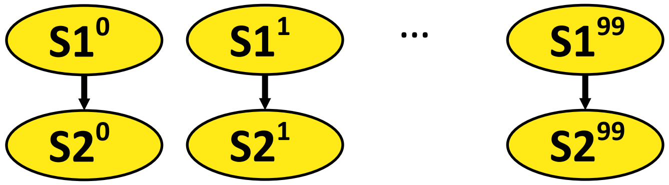
Iteration Space

- 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;
}
```

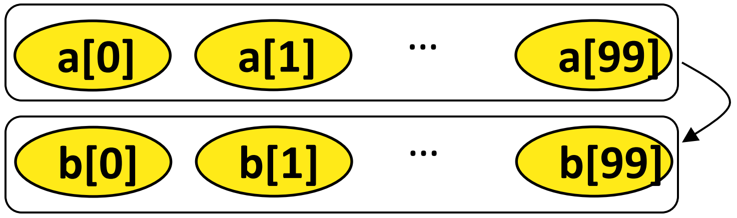




Multi-Loop Parallelism

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

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

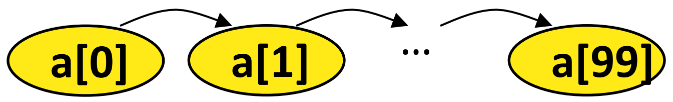


- Dependencies?
- How much parallelism is available?
- Given 4 processors, how much parallelism is possible?
- What parallelism is achievable with 50 processors?

Loops with Dependencies

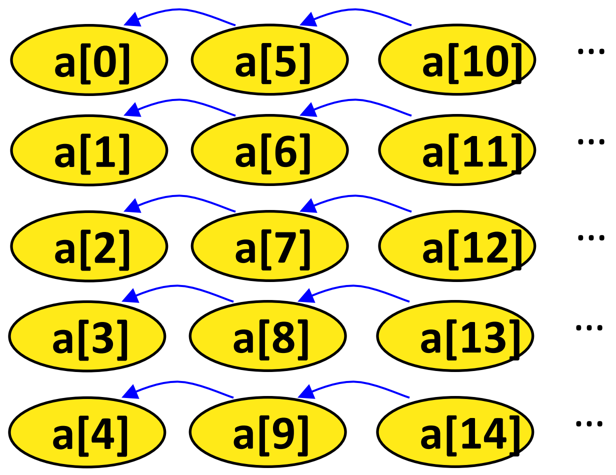
Case 1:

```
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

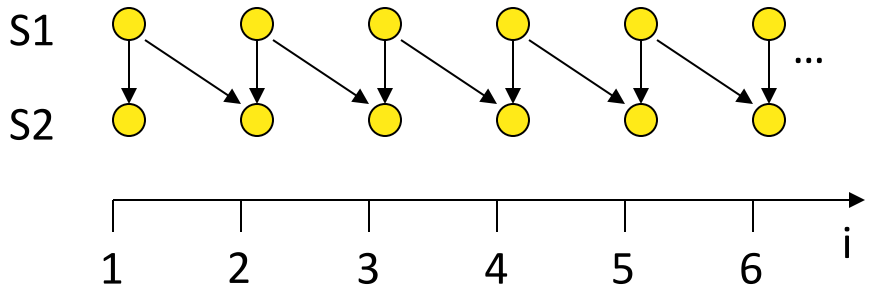
Loop Dependence Example

```
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 have 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

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

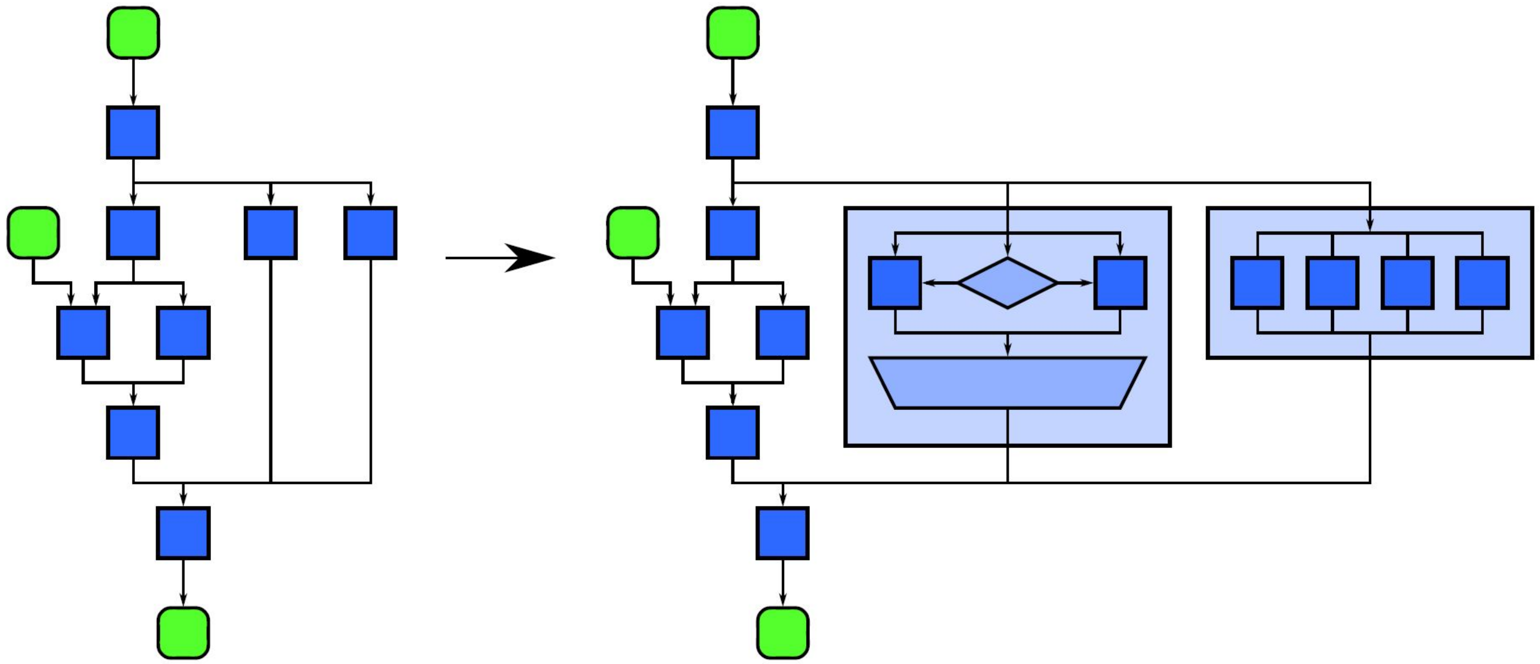
Parallel Patterns

- ❑ 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.

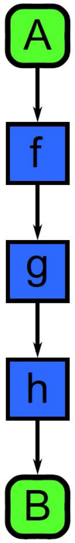
Serial Control Patterns

- 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

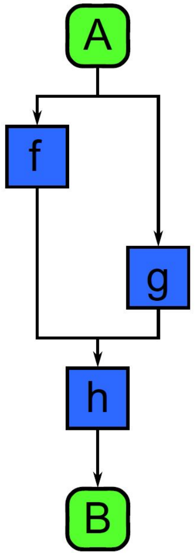
Serial Control Patterns: Sequence

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

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



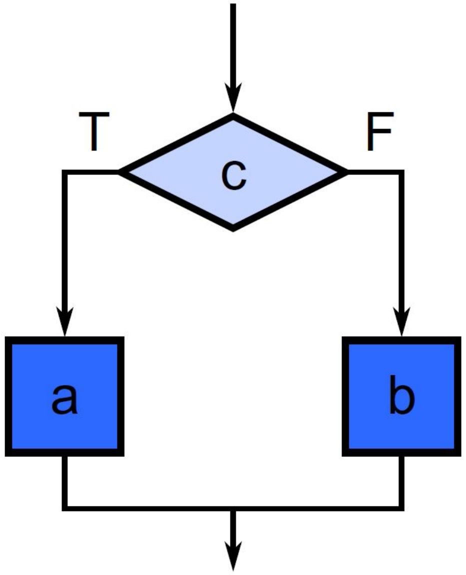
```
1   T = f(A);
2   S = g(A);
3   B = h(S,T);
```



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 only *a* or *b* is executed - never both

```
if (c) {
  a;
} else {
  b;
}
```

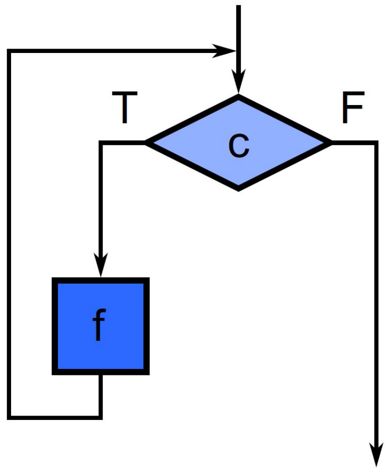


Serial Control Patterns: Iteration

- **Iteration**: a condition *c* is evaluated. If 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;
  a;
}
```

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



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

Parallel Control Patterns

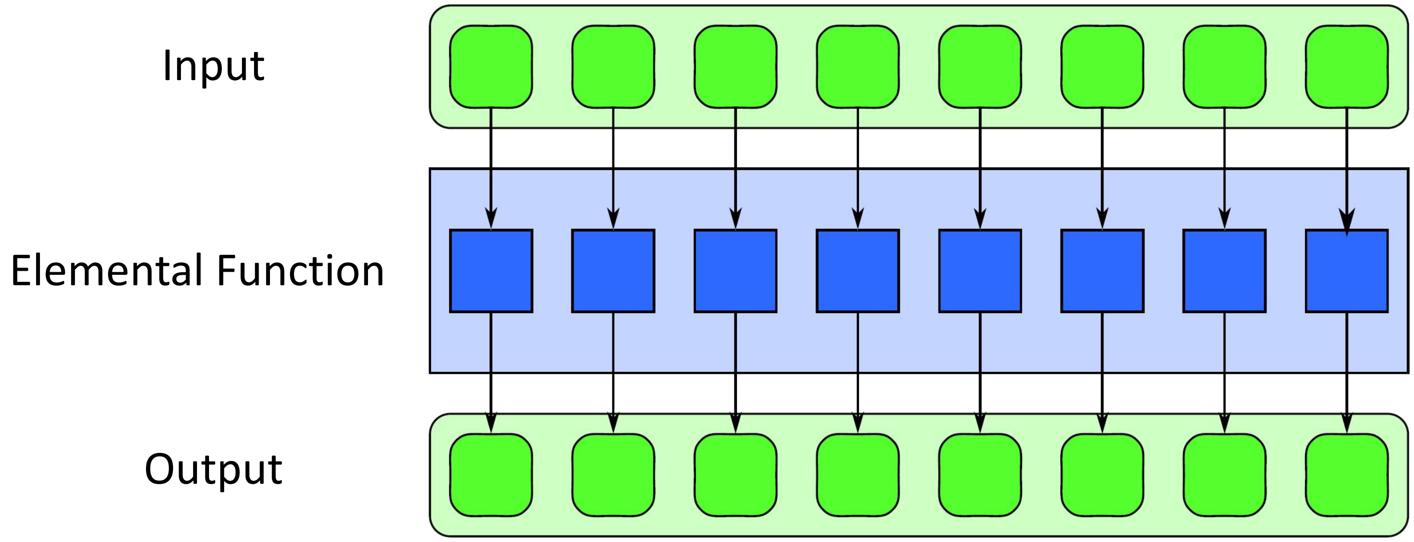
- Parallel control patterns extend serial control patterns
- Each parallel control pattern is related to at least one serial control pattern, but relaxes assumptions of serial control patterns
- Parallel control patterns: **fork-join**, **map**, **stencil**, **reduction**, **scan**, **recurrence**

Parallel Control Patterns: Fork-Join

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

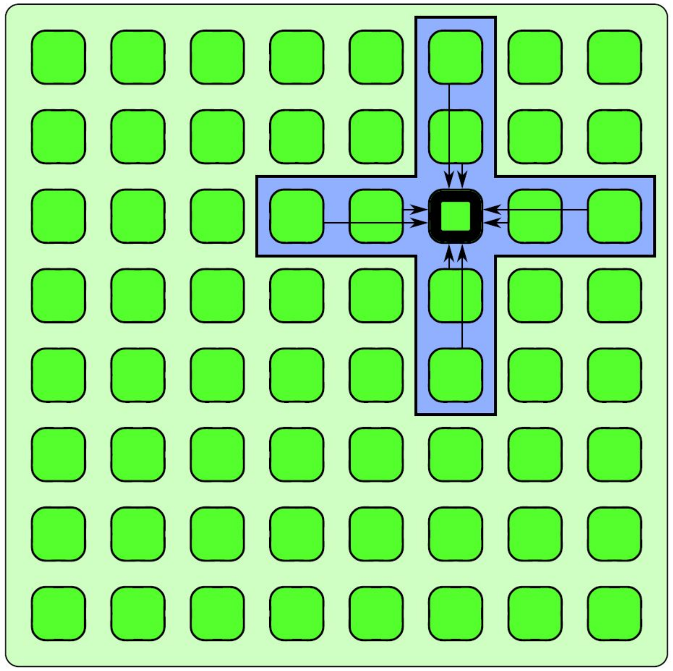
Parallel Control Patterns: Map

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



Parallel Control Patterns: Stencil

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

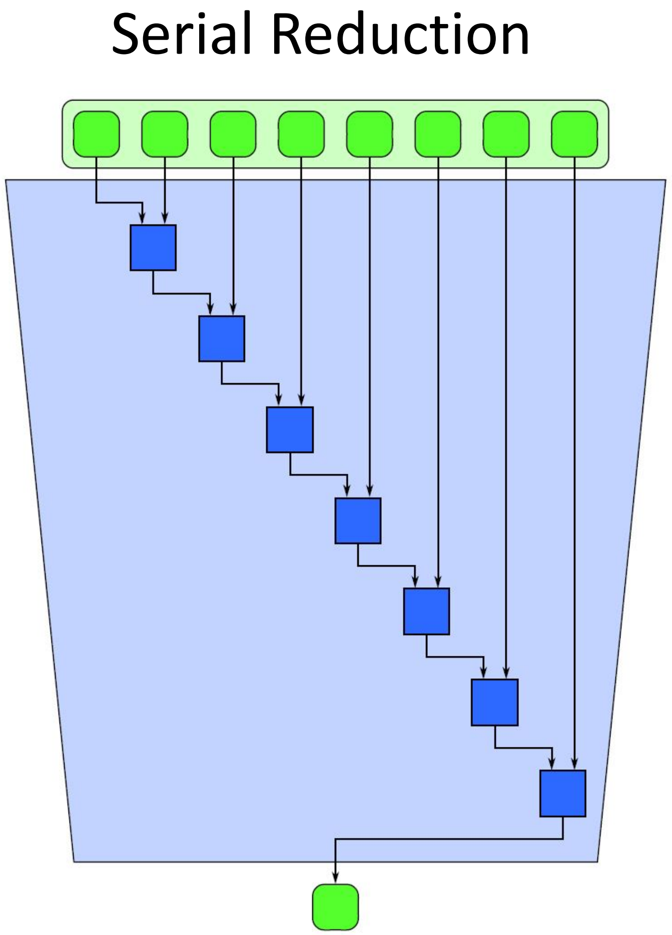


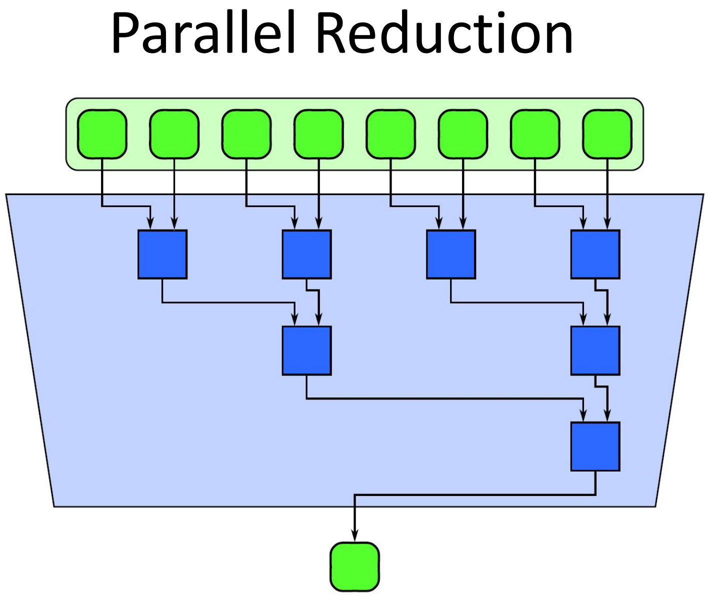
- Boundary conditions must be handled carefully in the stencil pattern
- See stencil lecture…

Parallel Control Patterns: Reduction

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

Parallel Control Patterns: Reduction



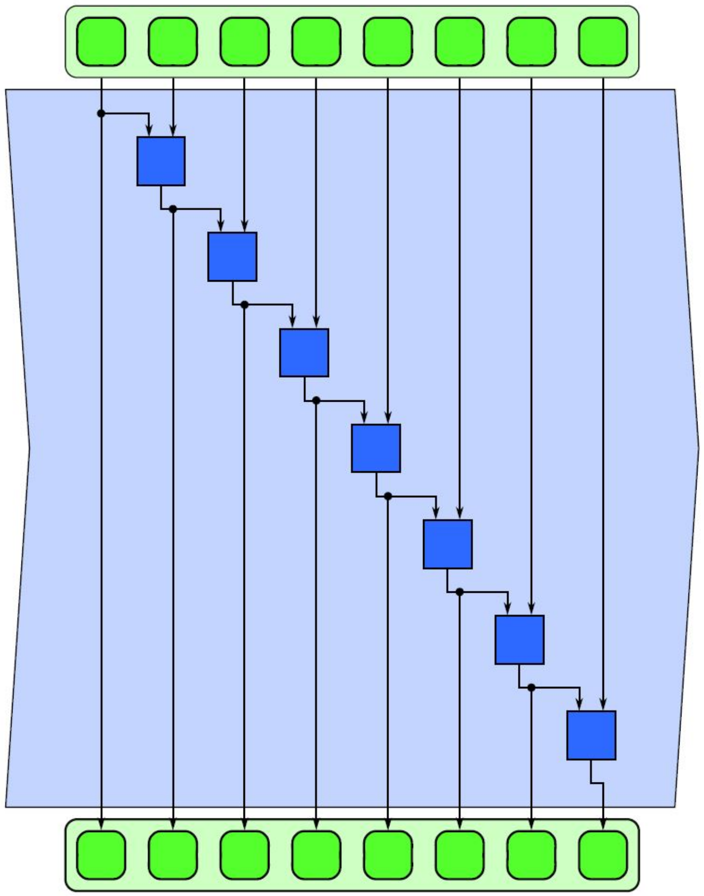


Parallel Control Patterns: Scan

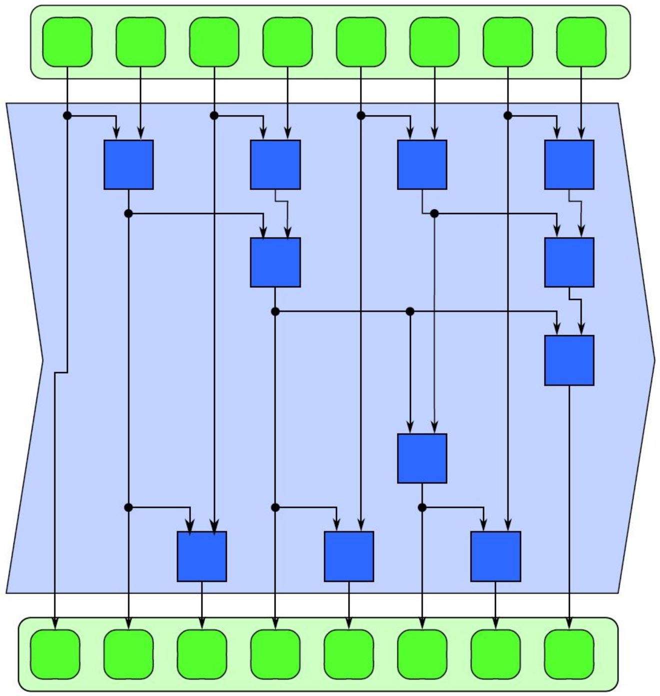
- **Scan**: computes all partial reduction of a collection
- For every output in a collection, a reduction of the input up to that point is computed
- If the function being used is associative, the scan can be parallelized
- Parallelizing a scan is not obvious at first, because of dependencies to previous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

Parallel Control Patterns: Scan

Serial Scan



Parallel Scan



Parallel Control Patterns: Recurrence

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