CIS 431/531
Intro to Parallel Computing

Parallel Patterns

Previously…

OpenMP

- Simple way of parallelizing code based on the fork-join model
- Loops, sections, tasks, etc.
- How to map/assign OpenMP threads to threads/cores/sockets
- Assignments
  - Calculate Pi in parallel (using a circle and Monte Carlo method)
  - Calculate a prefix sum in parallel

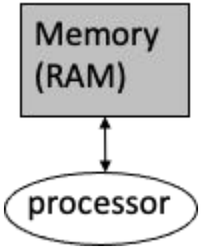
Today

- Dependencies
  - Loop carried dependency
- Parallel patterns
  - Recurring combination of task distribution and/or data access

Parallel Models

Sequential Models

- Random Access Machine (RAM) Model
- von Neumann model



Parallel Models

- A parallel computer is simply a collection of processors interconnected in some manner to coordinate activities and exchange data

These models are used as a general framework for describing and analyzing parallel algorithms

- Three common parallel models - directed acyclic graphs, shared-memory, and network

Directed Acyclic Graph

Captures data flow parallelism

Nodes represent operations/tasks to be performed

- Edges represent dependency or flow of data/results
- Nodes without any incoming edges - input
- Nodes without any outgoing edges - output

DAG represents the operations involved in the algorithm and constraints in the order of execution

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

Shared Memory Model

Parallel extension to the RAM model (PRAM)

- Memory size is infinite, number of processors is unbounded
- Processors communicate via the memory
- Each processor accesses memory in 1 cycle, and each instruction completes in 1 cycle

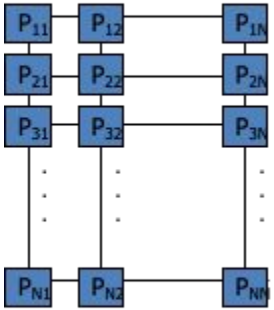
As with the RAM model, it neglects important practical properties such as memory/instruction latency and synchronization

Network

G = (N, E)

N are the processing nodes

E are bidirectional communication links



Each processor has its own memory

No shared memory (between the nodes)

Network operations may be synchronous or asynchronous and requires communication primitives

send (X, i)

receive(Y, j)

Captures the message passing model for algorithm design

Parallelism

Formally, the ability to execute different parts of the computation concurrently on different machines

Parallelism

- Reduces running/execution time
- Better resource utilization

What is being parallelized?

- Tasks - instructions, functions, etc.
- Data

Granularity

- Coarse-grained and fine-grained

Parallel Algorithms

"Recipe" for solving a problem on multiple processing elements

Standard steps for creating a parallel algorithm

- Identify work (e.g., instructions, data) that can be performed concurrently
- Partition the concurrent work on separate processing elements
- Properly manage input, output, and intermediate data
- Coordinate data accesses to to satisfy dependencies

Which step is the most difficult to handle?

Questions?

Dependency

Code 1
a = 1;
b = 2;
	Independent

Code 2 **a** = 1; b = **a** + 1;	Dependent True dependency (RAW, flow)
Code 3 **a** = 1; **a** = 2;	Dependent Output dependency (WAW)
Code 4 a = **b** + 1; **b** = 1;	Dependent Anti-dependency (WAR)

Dependency Graph

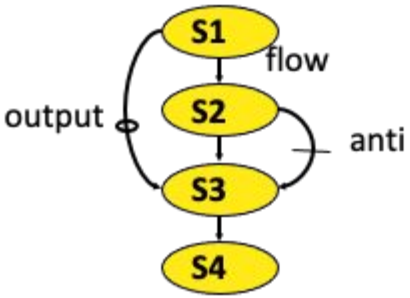
You can use a DAG to visually represent dependency

S1: a=1;

S2: b=a;

S3: a=b+1;

S4: c=a;



Which instructions can execute in parallel?

Dependency Graph

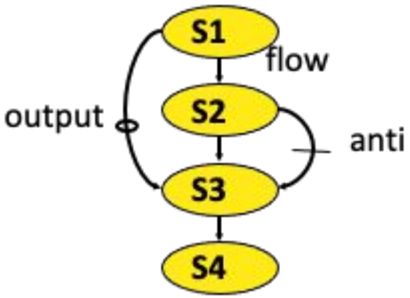
You can use a DAG to visually represent dependency

S1: a=1;

S2: b=a;

S3: a=b+1;

S4: c=a;



Which instructions can execute in parallel?

Dependency Graph

Can we determine this in a more systemic manner?

Yes, by comparing IN and OUT sets for each node

- IN - set of all memory locations (variables) that may be used in node S
- OUT - set of all memory locations (variables) that may be modified by node S

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

$$out(S_1) \cap in(S_2) \neq \varnothing \qquad S_1\ \delta\ S_2 \qquad \text{flow dependence}$$

$$in(S_1) \cap out(S_2) \neq \varnothing \qquad S_1\ \delta^{-1}\ S_2 \qquad \text{anti-dependence}$$

$$out(S_1) \cap out(S_2) \neq \varnothing \qquad S_1\ \delta^{0} S_2 \qquad \text{output dependence}$$

Dependency Graph

- IN - set of all memory locations (variables) that may be used in node S
- OUT - set of all memory locations (variables) that may be modified by node S

$$out(S_1) \cap in(S_2) \neq \varnothing \quad S_1\ \delta\ S_2 \quad \text{flow dependence}$$

$$in(S_1) \cap out(S_2) \neq \varnothing \quad S_1\ \delta^{-1}\ S_2 \quad \text{anti - dependence}$$

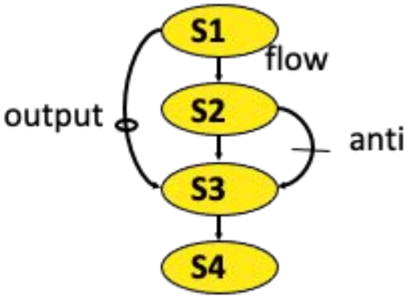
$$out(S_1) \cap out(S_2) \neq \varnothing \quad S_1\ \delta^{0}\ S_2 \quad \text{output dependence}$$

S1: a=1;

S2: b=a;

S3: a=b+1;

S4: c=a;



Identifying Parallelism

Significant amount of parallelism are often found in 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;
}
```

Dependency?

DOALL loop (*foreach* loop)

All iterations are independent of each other - all statements can be executed at the same time

Identifying Parallelism

What about here?

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

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

Are there any dependencies? What kind?

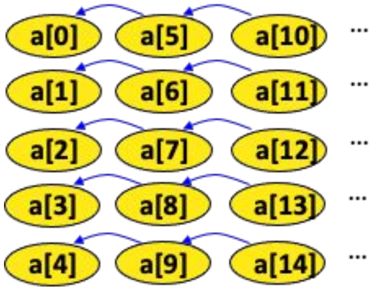
Identifying Parallelism

What about here?

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

a[0] = **a[5]** + 100;
a[5] = a[10] + 100;

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



Loop-carried Dependence

A **loop-carried** dependence occurs when there is a dependence between statements instances in two different iterations of a loop

Loop carried dependence can prevent loop iterations from being parallelized using DOALL

Dependence is **lexically forward** if source comes before the target, or **lexically backward** otherwise

Unrolling the loop can help figure this out

Loop-carried Dependence

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

Dependency?

Between a[0] and a[10], between a[10] and a[20], etc.

Between a[1] and a[11], between a[11] and a[21], etc.

Is it possible to parallelize this loop? How?

Loop-carried Dependence

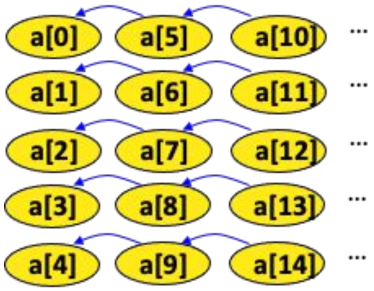
for (i=0; i<100; i++)

a[i+10] = f(a[i]);

Similar to:

for (i=5; i<100; i++)

a[i-5] = a[i] + 100;



Dependency?

Between a[0] and a[10], between a[10] and a[20], etc.

Between a[1] and a[11], between a[11] and a[21], etc.

Is it possible to parallelize this loop? How?

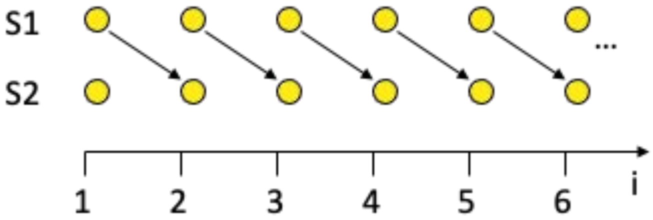
Loop-carried Dependence

```
for (i=1; i<100; i++)
    S1: a[i] = ...;
    S2: ... = a[i - 1];
```

Is it possible to parallelize this loop? How?

Loop-carried Dependence

```
for (i=1; i<100; i++)
    S1: a[i] = ...;
    S2: ... = a[i - 1];
```

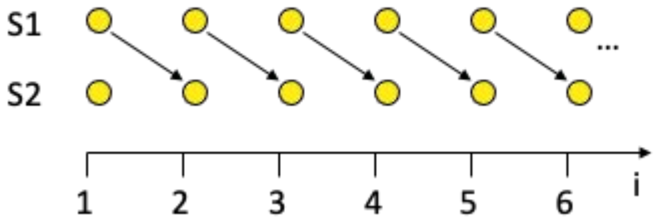


Is it possible to parallelize this loop? How?

Software pipelining

Loop-carried Dependence

```
for (i=1; i<100; i++)
    S1: a[i] = ...;
    S2: ... = a[i - 1];
```




```
for (i=1; i<100; i+=4)
    S1: a[1] = ...;
    S2: ... = a[0];
    S1: a[2] = ...;
    S2: ... = a[1];
    S1: a[3] = ...;
    S2: ... = a[2];
    S1: a[4] = ...;
    S2: ... = a[3];
```

⇒

```
for (i=1; i<100; i+=4)
    S1: a[1] = ...;
    S1: a[2] = ...;
    S1: a[3] = ...;
    S1: a[4] = ...;
    S2: ... = a[0];
    S2: ... = a[1];
    S2: ... = a[2];
    S2: ... = a[3];
```

Loop-carried Dependence

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

Dependency?

Loop-carried Dependence

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

Dependency?

Loop **independence** on i

Loop-carried **dependency** on j -> outer loop can be parallelized

Synchronization

How is parallelism achieved when there are dependencies?

A way to force ordering of tasks (on different processors/cores) is required

Use synchronization mechanisms

- Barriers, locks, semaphores

Questions?

Parallel Patterns

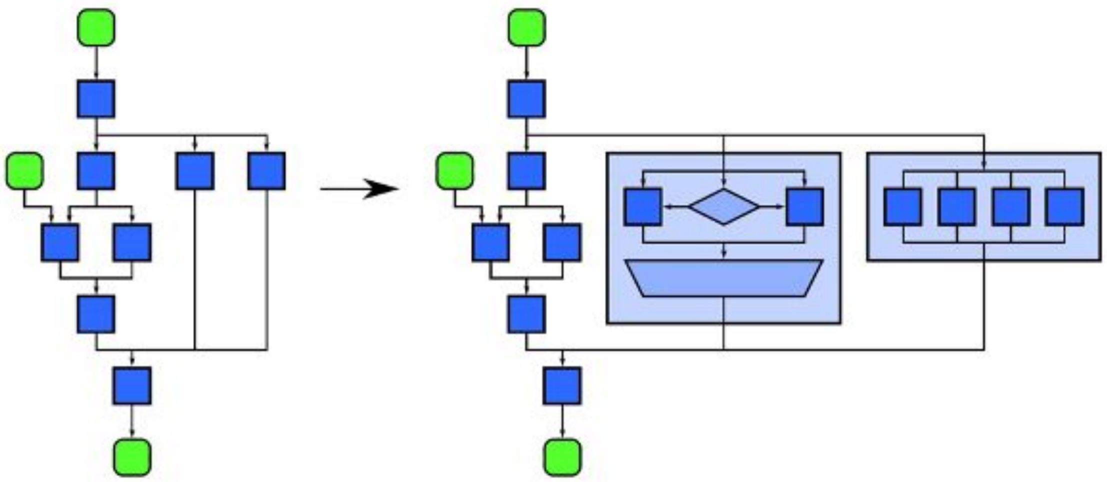
- A recurring combination of task distribution and data access
  - Nesting patterns
  - Control patterns
  - Data management patterns
  - Others
- Some programming models are based on specific parallel patterns

Nesting

Ability to hierarchically compose patterns

Can be both serial and parallel

Any task block on the left can be replaced by another pattern with the same input/output dependency



Control

Serial

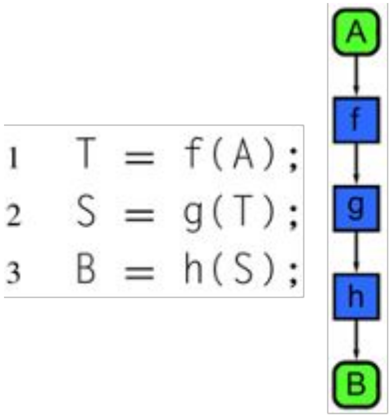
- Sequence, selection, iteration, and recursion

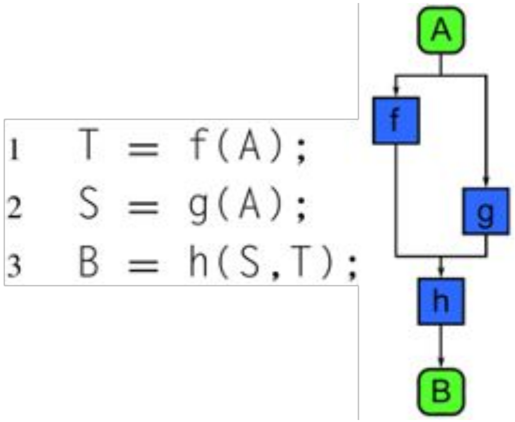
Parallel

- Fork-join, map, stencil, reduction, scan, recurrence

Sequence

Ordered list of tasks

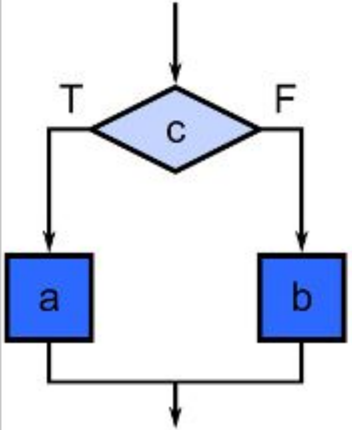




Selection

Condition *c* is first evaluated, and either task *a* **or** *b* is executed depending on *c*

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

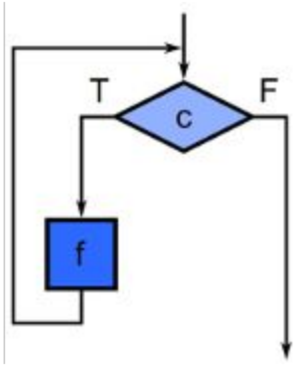


Iteration

Condition *c* is evaluated, and if true, *a* is executed, and then the process repeats until *c* is false

```
for (i = 0; i < n;
  a;
}
```

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



Recursion

Dynamic form of nesting, where a function call itself repeatedly

Tail recursion is a special recursion that can be converted into iteration - easier for the compiler to optimize and/or parallelize

Parallel Control Patterns

Parallel control patterns extend serial ones (i.e., each is related to at least one serial one)

Types

Fork-join, map, stencil, reduction, scan, recurrence

Fork-join

Allows control to “fork” into multiple parallel flows, which later “joins”

- Cilk Plus implements this with *spawn* and *sync*
- OpenMP uses #pragma parallel to create parallel regions

A “join” is different from a barrier

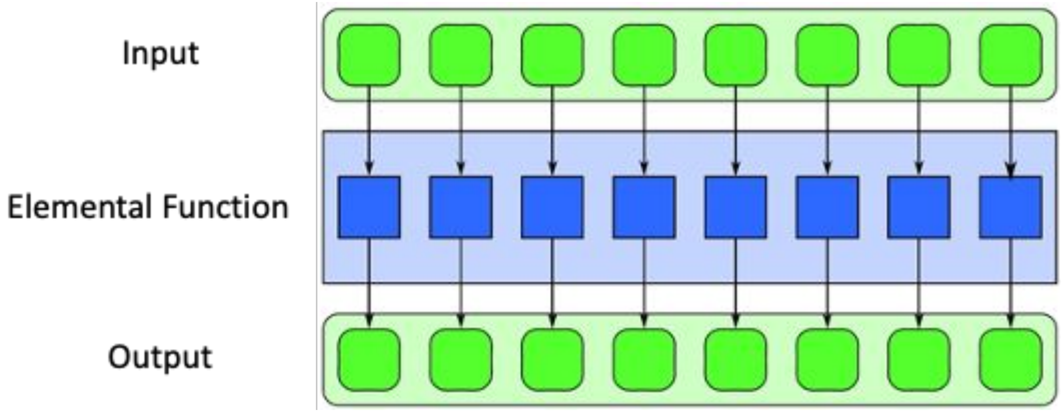
- Join - only one thread continues after synchronizing at the join
- Barrier - all threads continue after synchronizing at the barrier

Map

Performs a function/task over every element of a collection

Map replicates a serial iteration pattern, where each iteration is independent of others

Replicated function is also referred to as an elemental function



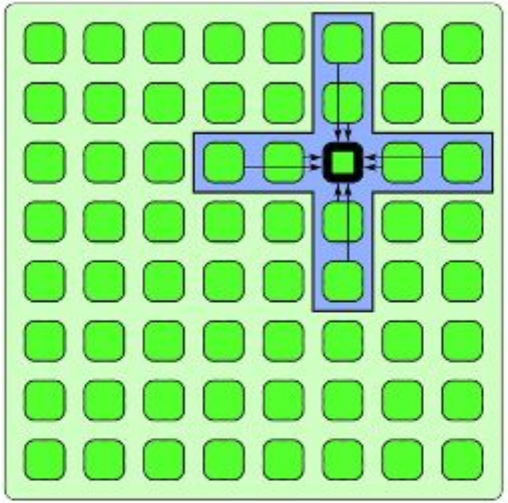
Stencil

Elemental function accesses a set of "neighbors"

Generalization of a map

Often combined with iteration

Boundary condition must be handled carefully/differently



Reduction

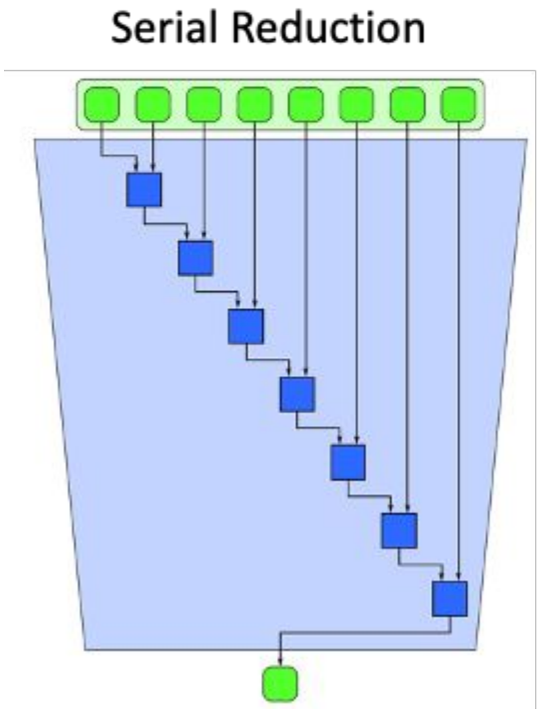
Combines every element in a collection using an **associative** "combiner" function

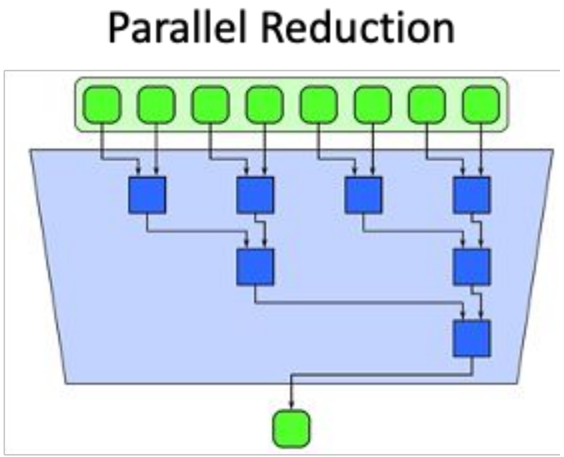
The associativity allows different ordering of the combination (and therefore allow parallelization)

Examples are:

Add, multiply, max, min, AND, OR, etc.

Reduction





Scan

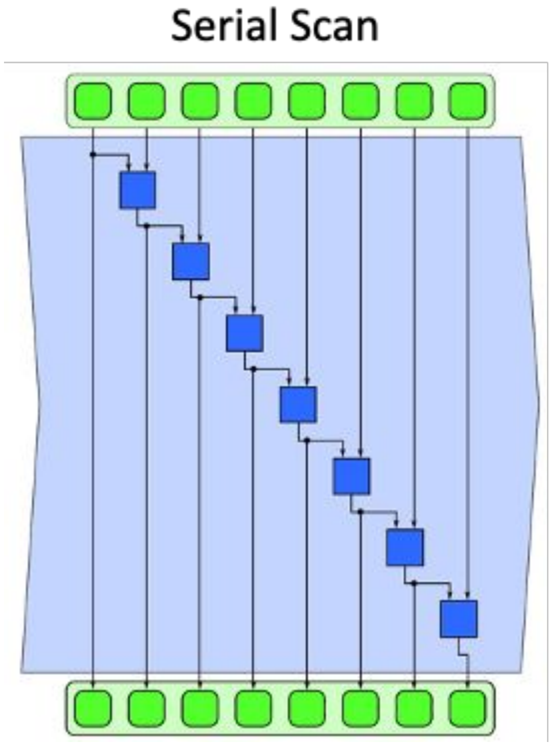
Computes partial reduction of a collection

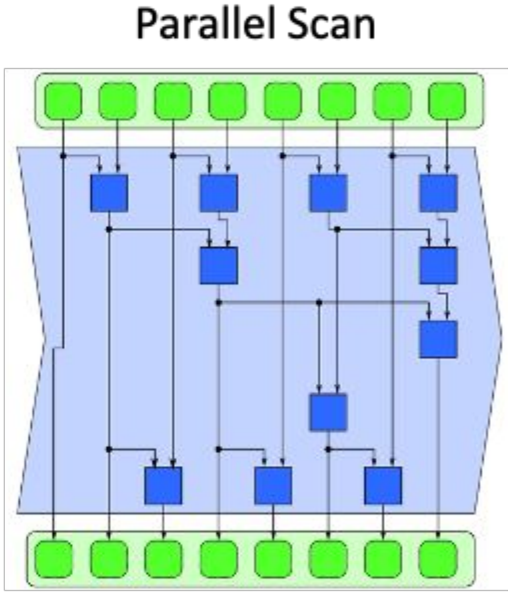
For every output in a collection, a reduction of the input **up to that point** is computed

If the function is associative, scan can be parallelized
Example:

Prefix sum

Scan





Recurrence

More complex version of map, where the loop iteration can depend on one another

Similar to map, but elements can use outputs of **adjacent** elements as inputs

Recurrence requires serial ordering of dependent elements

Example:

$$T_1 = 1$$

$$T_n = T_{n-1} + 1 \text{ (for } n >= 2)$$

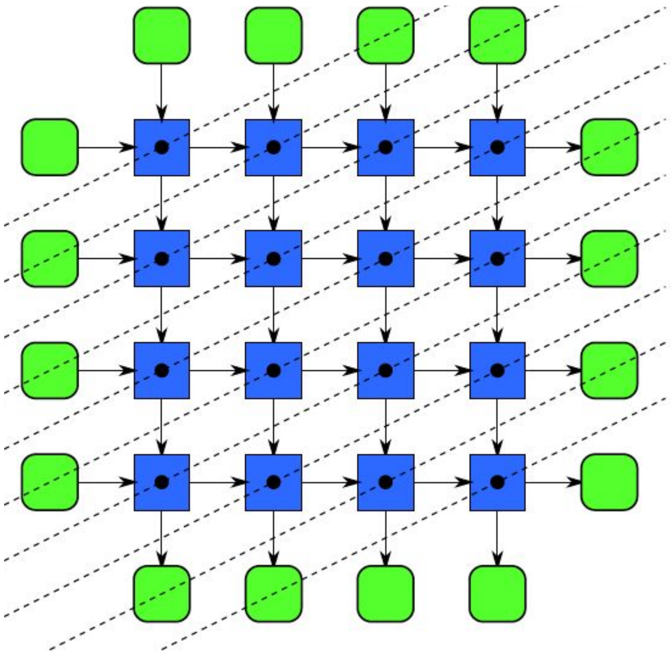
Recurrences are difficult to parallelize, but not necessarily impossible

Recurrence

```
for(i = 0; i < w; i++) {
  for(j = 0; j < h; j++) {
    b[i][j] = f(b[i - 1][j], b[i][j - 1], a[i][j])
  }
}
```

How would you parallelize this?

Recurrence



Data Management

Serial

- Random read/write, stack, heap, objects

Parallel

- Pack, pipeline, geometric decomposition, gather, scatter

Random read/write

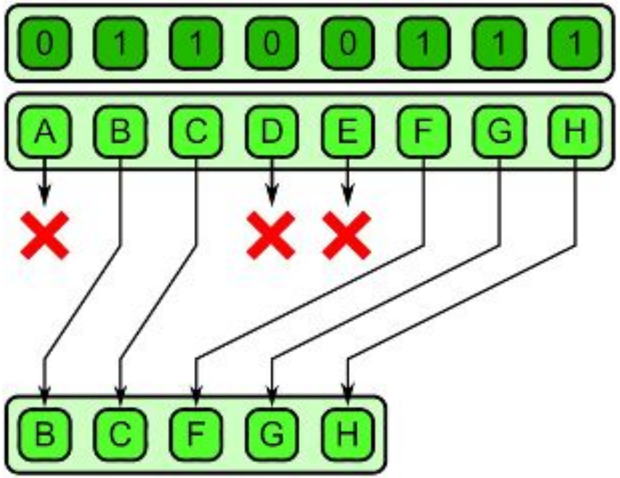
Memory locations are indexed with addresses (i.e., pointers)

Aliasing (uncertainty of two pointer referring to the same object) can cause problems when parallelizing

Pack

Eliminates unused space between elements in a collection

Unpack does the opposite

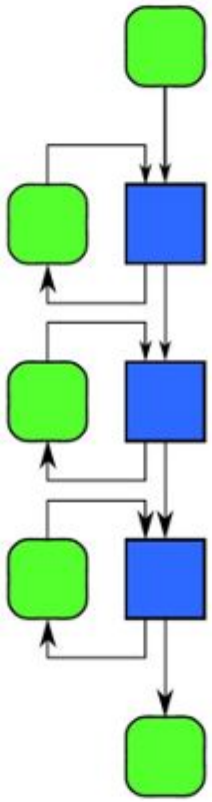


Pipeline

Connects data in a producer-consumer manner

Can be linear (basic) or can be in a DAG form

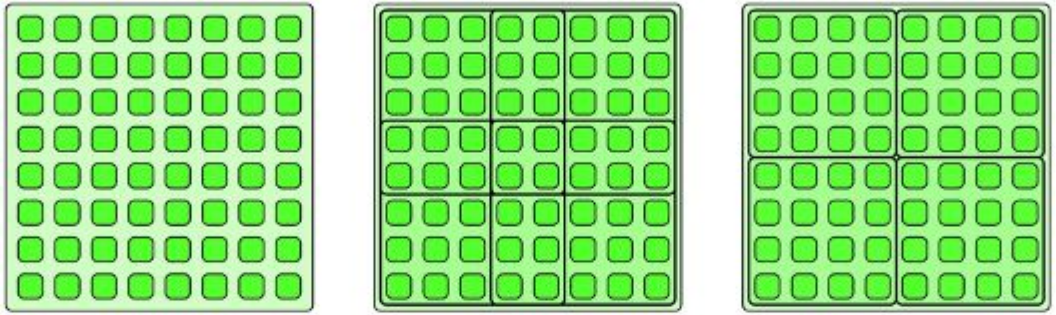
Typically used with other patterns (to increase parallelism)



Geometric Decomposition

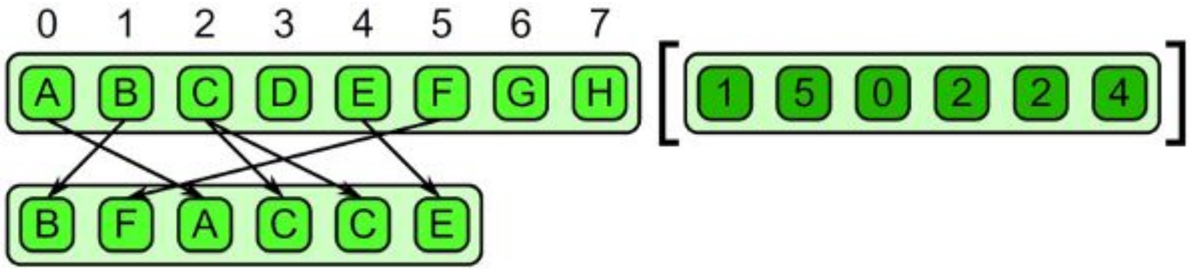
Arranges data into a subcollections

Can be overlapping or non-overlapping



Gather

Gather reads in a collection of data using a given collection of indices



Scatter

Inverse of gather

Race condition can occur when we have two writes to the same location