EE382N (20): Computer Architecture - Parallelism and Locality

Lecture 10 – Parallelism in Software I

Mattan Erez

The University of Texas at Austin

Credits

- Most of the slides courtesy Dr. Rodric Rabbah (IBM)
  - Taken from 6.189 IAP taught at MIT in 2007
- Parallel Scan slides courtesy David Kirk (NVIDIA) and Wen-Mei Hwu (UIUC)
  - Taken from EE493-AI taught at UIUC in Sprig 2007

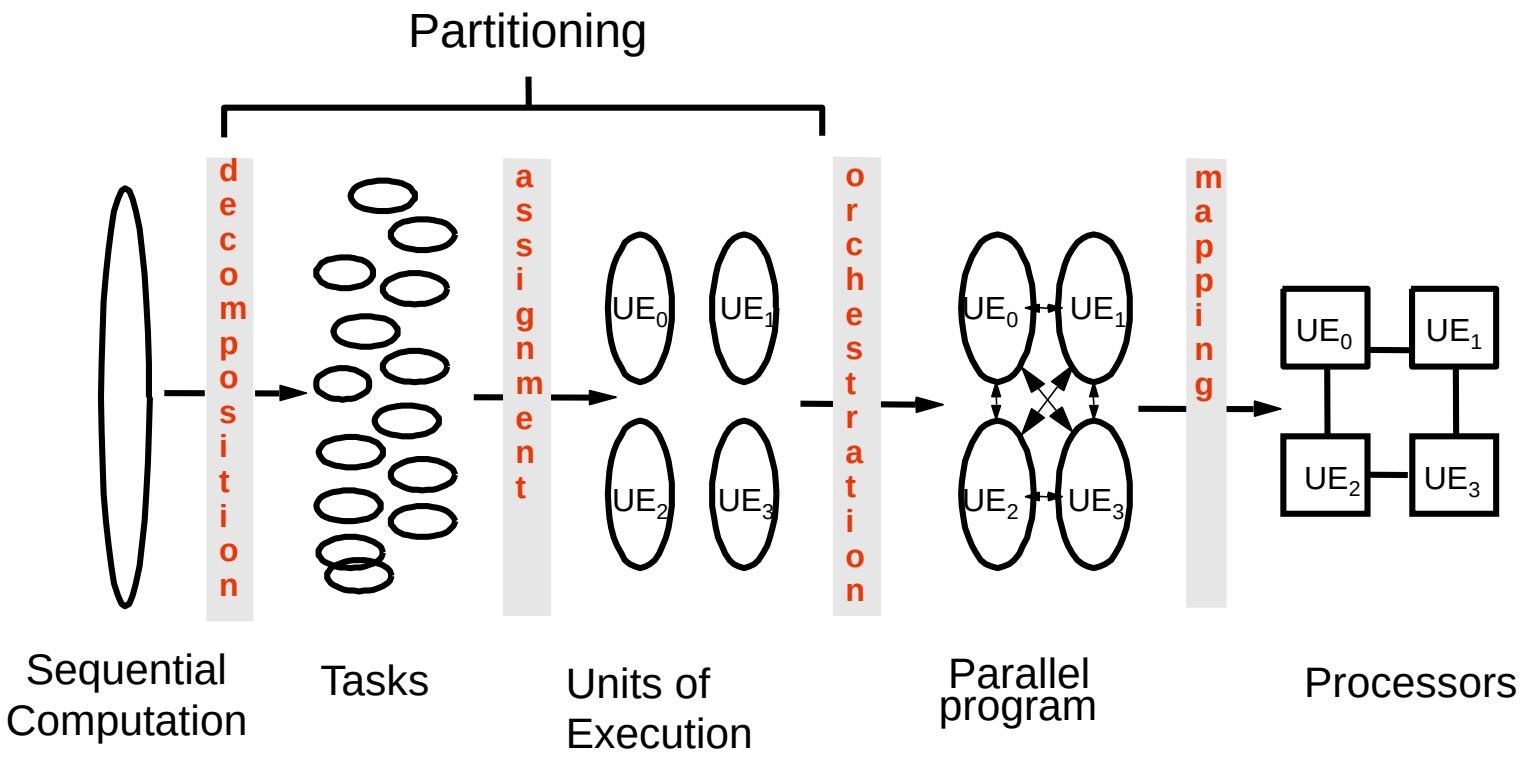
Parallel programming from scratch

- Start with an algorithm
  - Formal representation of problem solution
  - ***Sequence*** of steps
- Make sure there is parallelism
  - In each algorithm step
  - Minimize synchronization points
- Don't forget locality
  - Communication is costly
    - Performance, Energy, System cost
- More often start with existing sequential code

Reengineering for Parallelism

- Define a testing protocol
- Identify program hot spots: where is most of the time spent?
  - Look at code
  - Use profiling tools
- Parallelization
  - Start with hot spots first
  - Make sequences of small changes, each followed by testing
  - Patterns provide guidance

4 Common Steps to Creating a Parallel Program



1. Decomposition

- Identify concurrency and decide at what level to exploit it

- Break up computation into tasks to be divided among processes
  - Tasks may become available dynamically
  - Number of tasks may vary with time

- Enough tasks to keep processors busy
  - Number of tasks available at a time is upper bound on achievable speedup

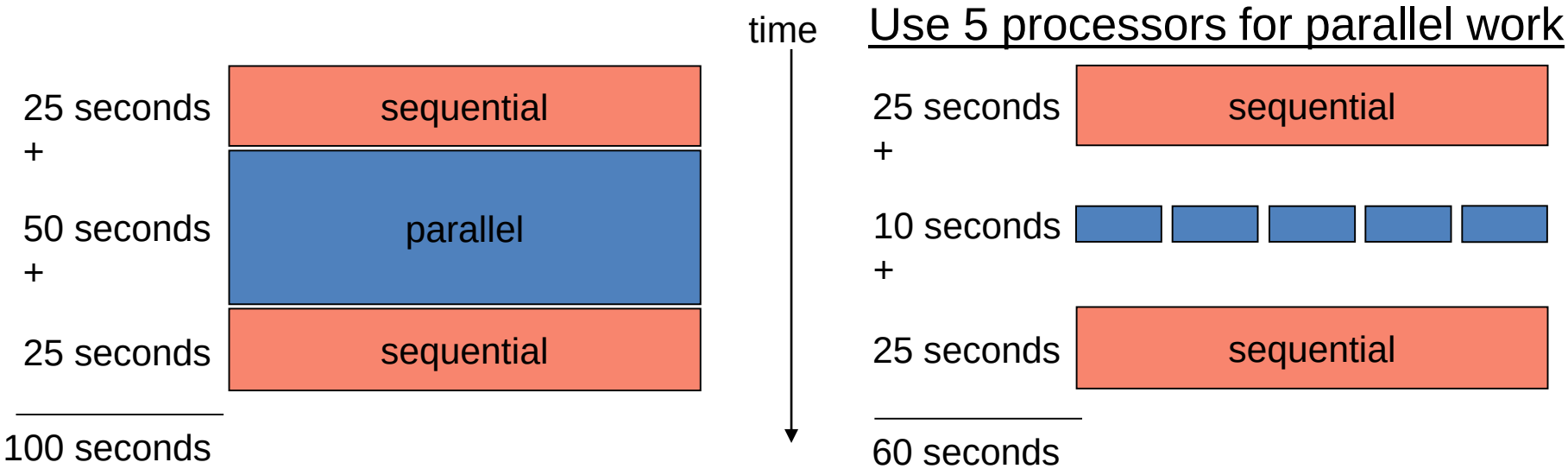
Main consideration: coverage and Amdahl's Law

Coverage

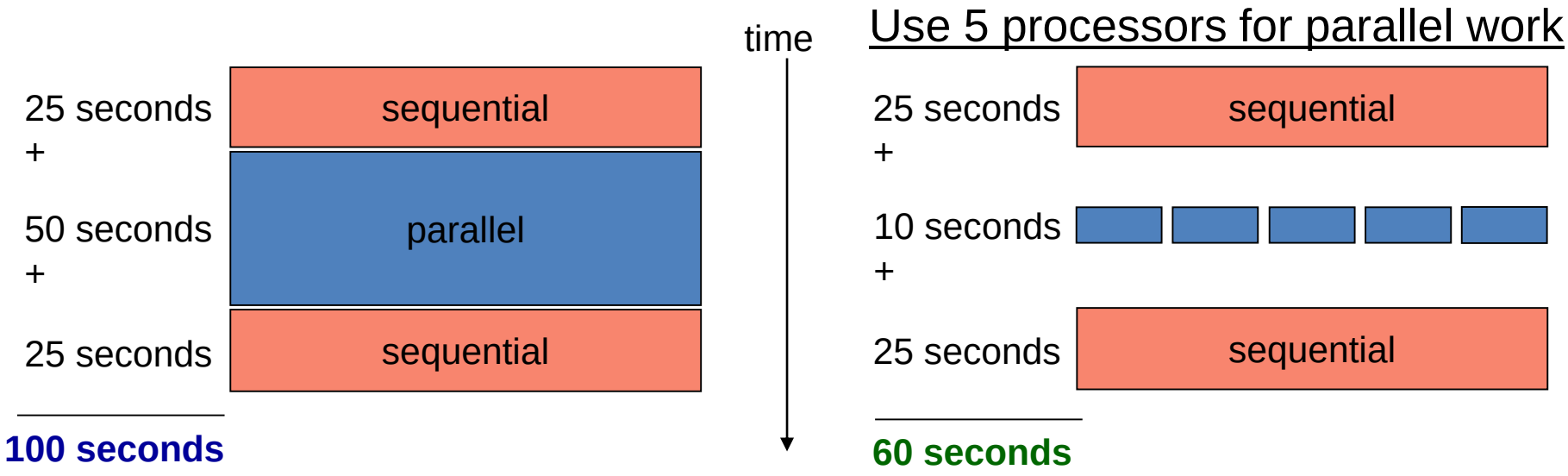
- **Amdahl's Law**: *The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.*
  - Demonstration of the law of diminishing returns

Amdahl's Law

- Potential program speedup is defined by the fraction of code that can be parallelized



Amdahl's Law



- Speedup= old running time / new running time
  = 100 seconds / 60 seconds
  = 1.67
  (parallel version is 1.67 times faster)

Amdahl's Law

- p = fraction of work that can be parallelized
- n = the number of processor

$$speedup = \frac{\text{old running time}}{\text{new running time}}$$

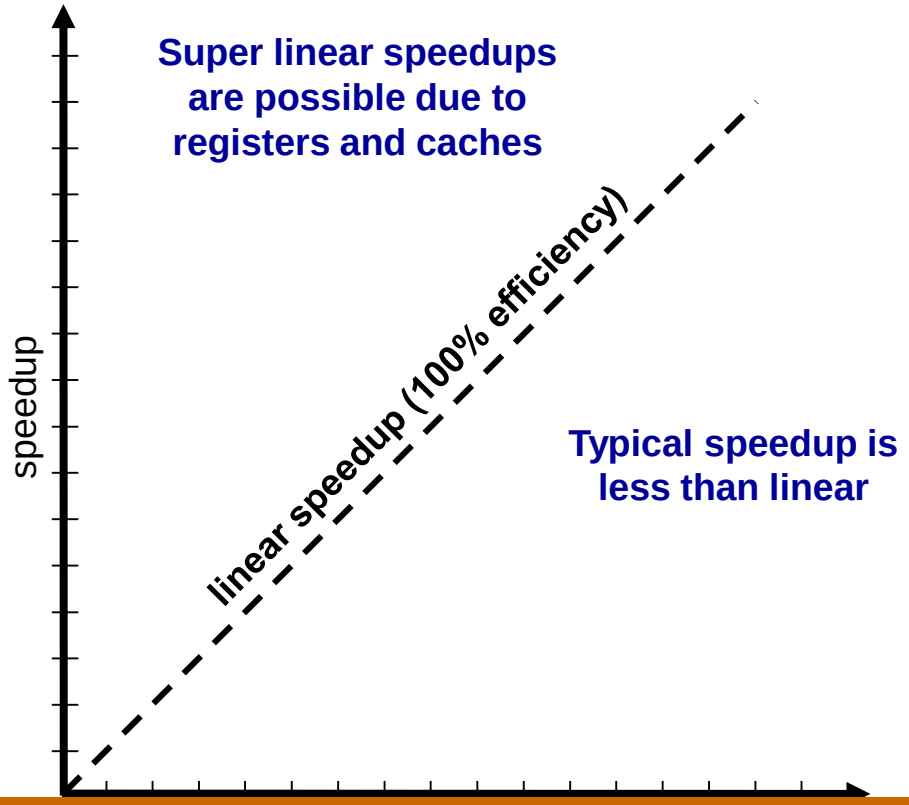
$$= \frac{1}{(1-p) + \frac{p}{n}}$$

fraction of time to complete sequential work

fraction of time to complete parallel work

Implications of Amdahl's Law

- Speedup tends to $\frac{1}{1-p}$ as number of processors tends to infinity



Parallelism only worthwhile when it dominates execution

Why is speedup less than linear?

- Synchronization
- Communication
- Load imbalance
- More work to do
  - Often, parallel algorithm has more work (why?)
  - Decomposition may create repeated work (why?)

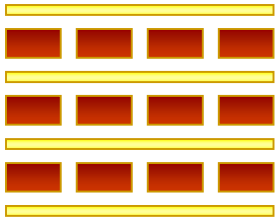
2. Assignment

- Specify mechanism to divide work among PEs
  - Balance work and reduce communication
- Structured approaches usually work well
  - Code inspection or understanding of application
  - Well-known design patterns
- As programmers, we worry about partitioning first
  - Independent of architecture or programming model?
  - Complexity often affects decisions
  - Architectural model affects decisions

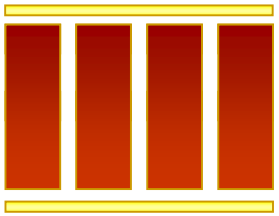
Main considerations: granularity and locality

Fine vs. Coarse Granularity

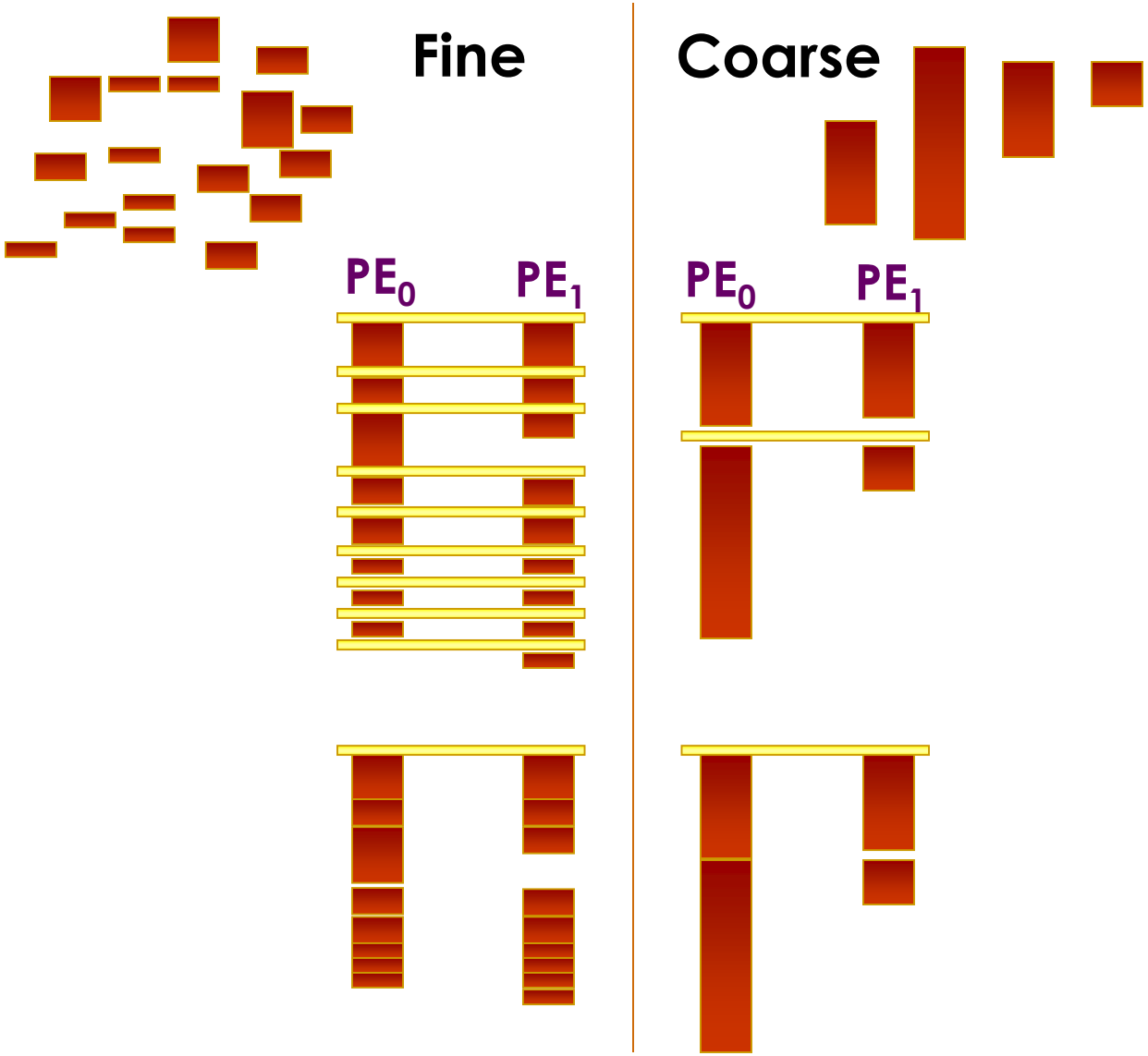
- Fine-grain Parallelism
  - Low computation to communication ratio
  - Small amounts of computational work between communication stages
  - High communication overhead
    - Potential HW assist



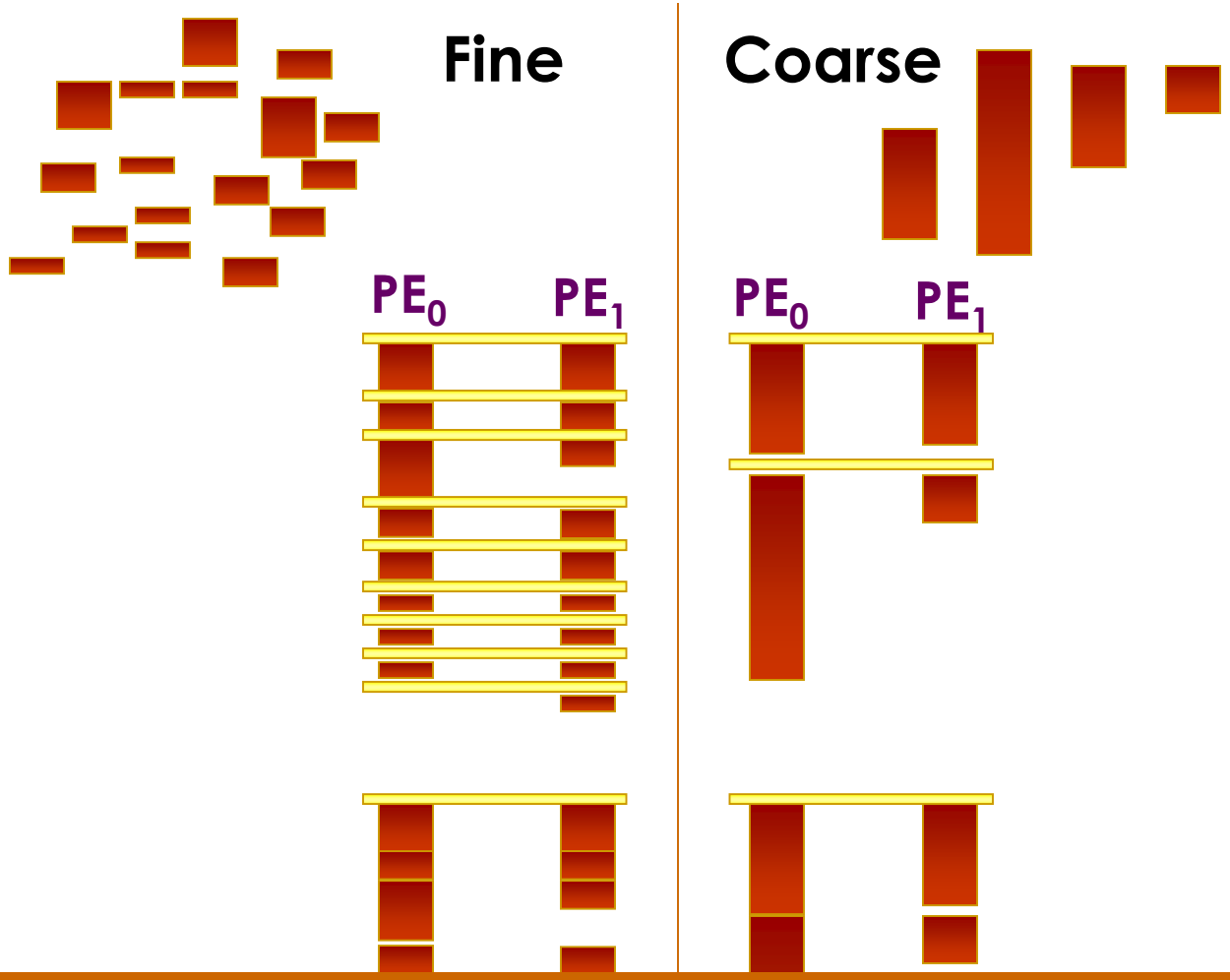
- Coarse-grain Parallelism
  - High computation to communication ratio
  - Large amounts of computational work between communication events
  - Harder to load balance efficiently



Load Balancing vs. Synchronization



Load Balancing vs. Synchronization



Expensive sync → coarse granularity

Few units of exec + time disparity → fine granularity

3+4. Orchestration and Mapping

- Computation and communication concurrency
- Preserve locality of data
- Schedule tasks to satisfy dependences early
- Survey available mechanisms on target system

Main considerations: locality, parallelism, mechanisms (efficiency and dangers)

Parallel Programming by Pattern

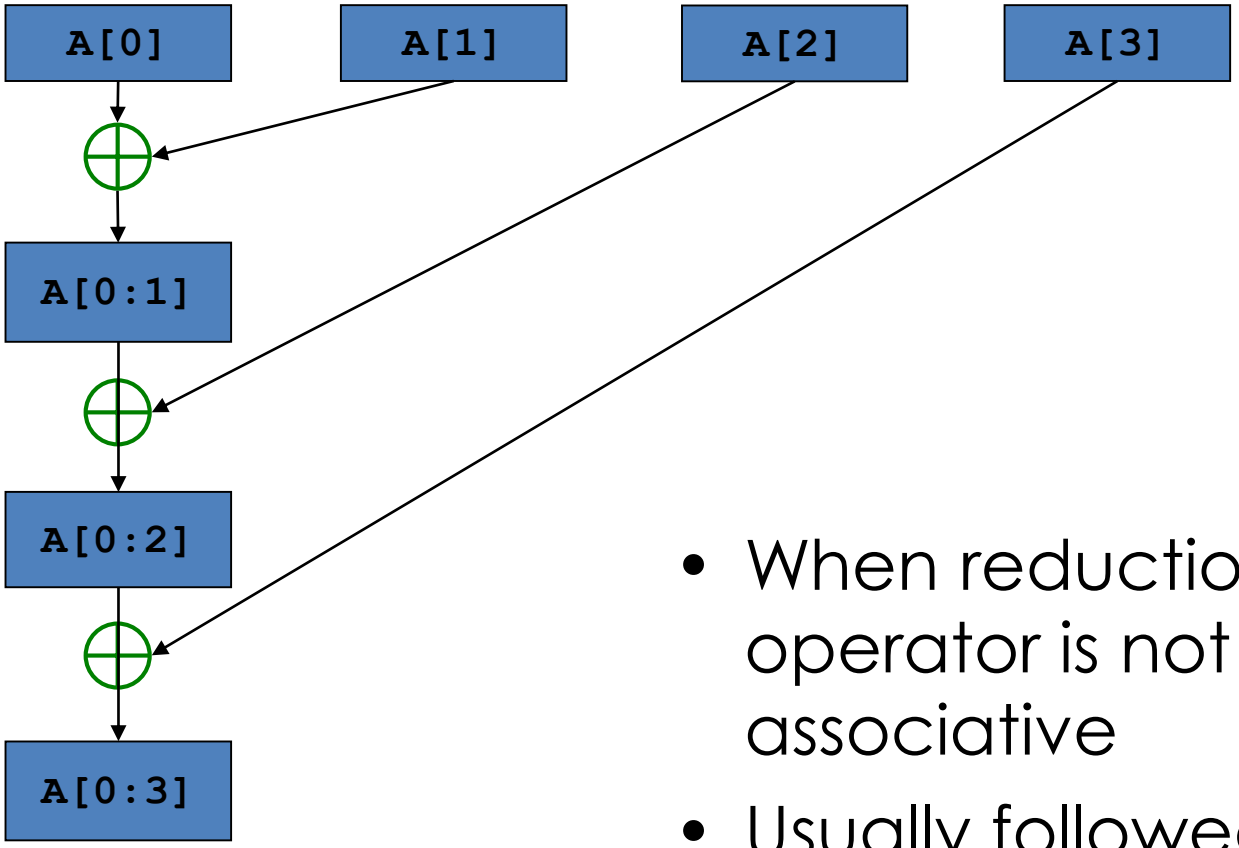
- Provides a cookbook to systematically guide programmers
  - Decompose, Assign, Orchestrate, Map
  - Can lead to high quality solutions in some domains
- Provide common vocabulary to the programming community
  - Each pattern has a name, providing a vocabulary for discussing solutions
- Helps with software reusability, malleability, and modularity
  - Written in prescribed format to allow the reader to quickly understand the solution and its context
- Otherwise, too difficult for programmers, and software will not fully exploit parallel hardware

GPU example pattern: reductions and scans

Reductions

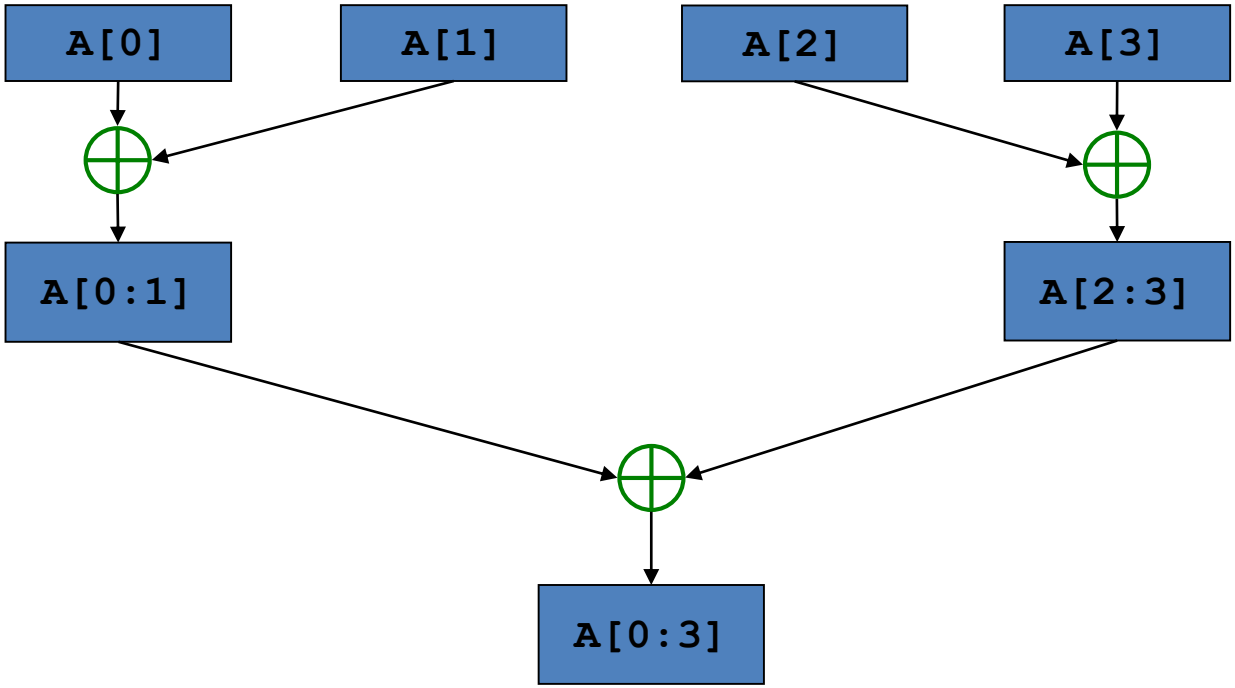
- Many to one
- Many to many
  - Simply multiple reductions
    - Also known as scatter-add and subset of parallel prefix sums
- Use
  - Histograms
  - Superposition
    - Physical properties

Serial Reduction



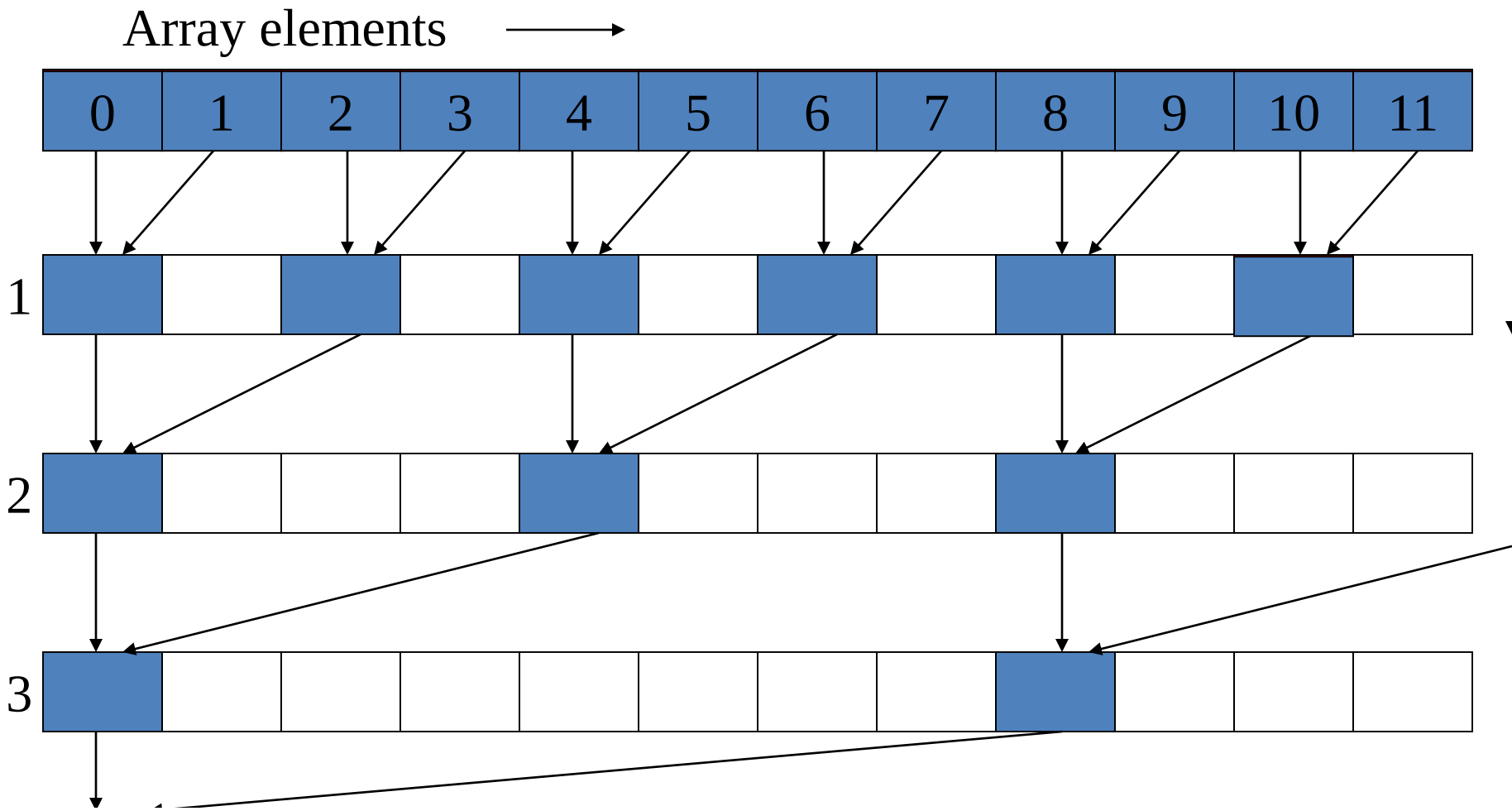
- When reduction operator is not associative
- Usually followed by a broadcast of result

Tree-based Reduction

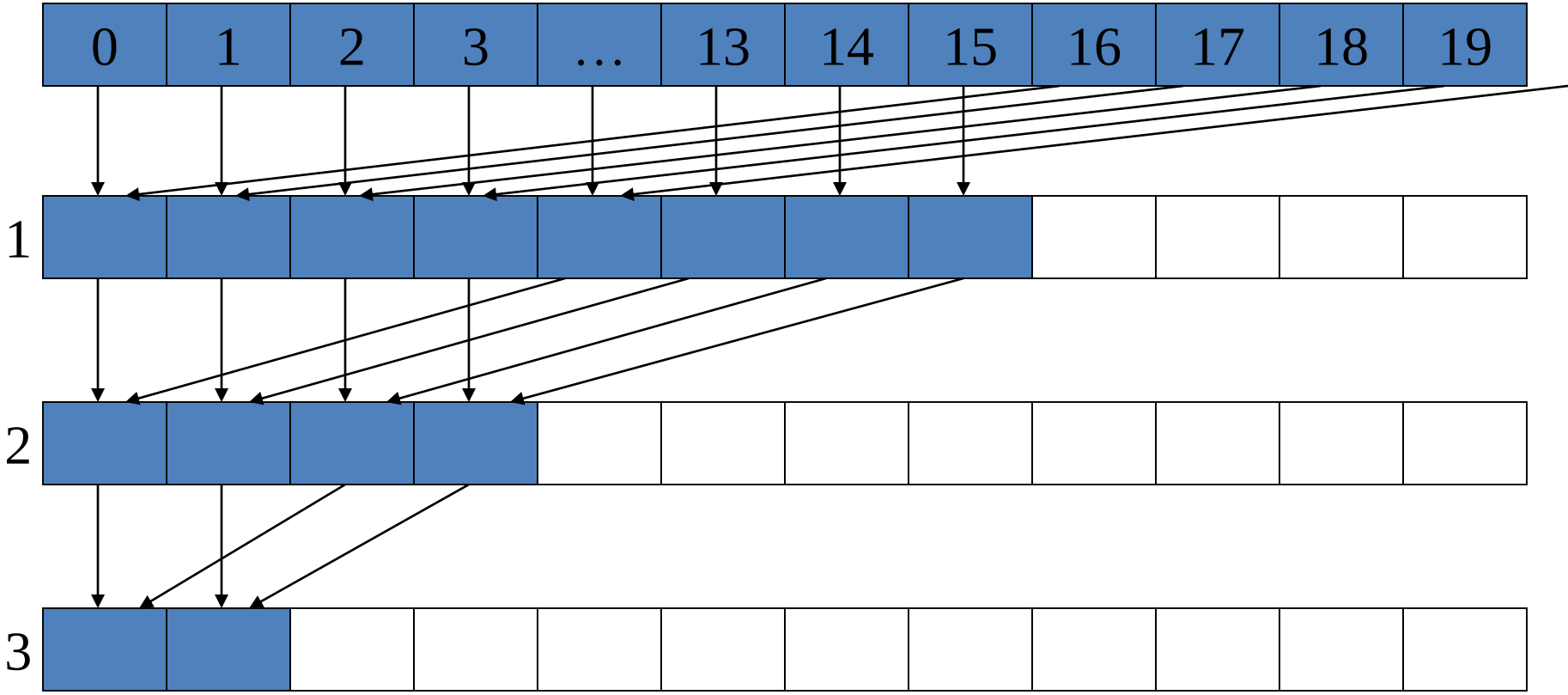


- n steps for 2^n units of execution
- When reduction operator is associative
- Especially attractive when only one task needs result

Vector Reduction with Bank Conflicts

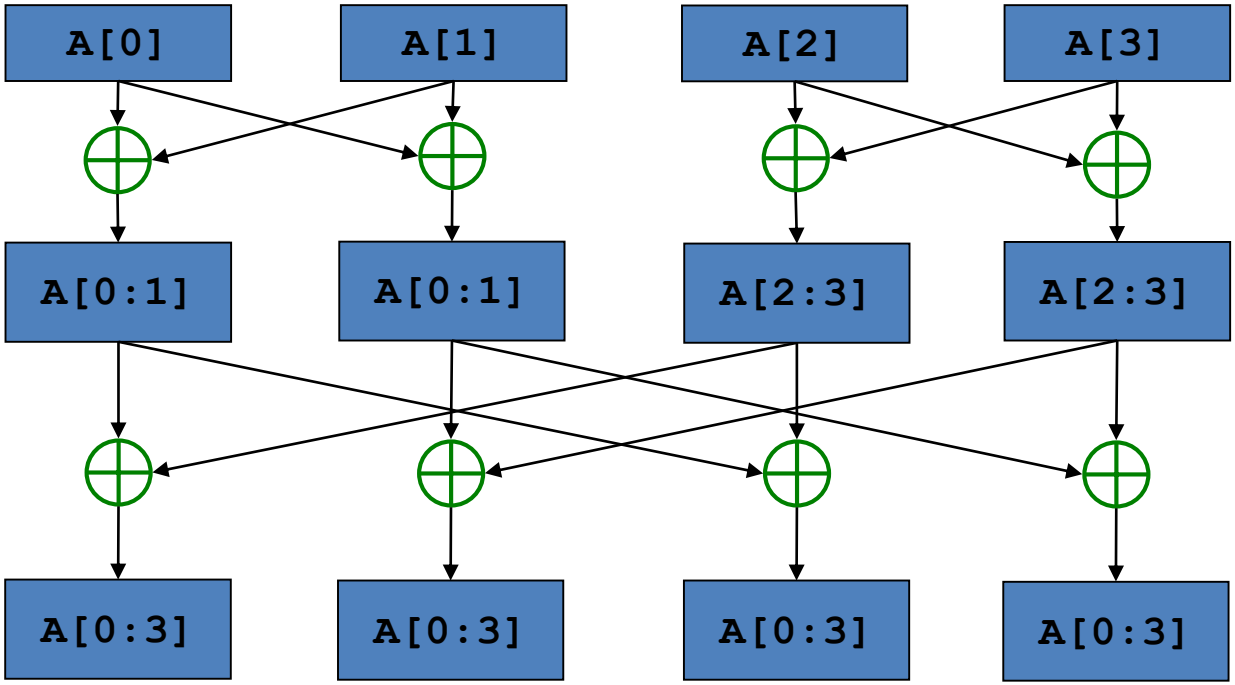


No Bank Conflicts



What else did optimized implementation gain?

Recursive-doubling Reduction



- n steps for 2^n units of execution
- If all units of execution need the result of the reduction

Recursive-doubling Reduction

- Better than tree-based approach with broadcast
  - Each units of execution has a copy of the reduced value at the end of n steps
  - In tree-based approach with broadcast
    - Reduction takes *n* steps
    - Broadcast cannot begin until reduction is complete
    - Broadcast can take *n* steps (architecture dependent)

Parallel Prefix Sum (Scan)

- Definition:

  The all-prefix-sums operation takes a binary associative operator $\oplus$ with identity I, and an array of n elements

  $$[a_0, a_1, \ldots, a_{n-1}]$$

  and returns the ordered set

  $$[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].$$

- Example:
  if $\oplus$ is addition, then scan on the se

  [3 1 7 0 4 1 6 3]

  returns the set

  [0 3 4 11 11 15 16 22]

Exclusive scan: last input element is not included in the result

(From Blelloch, 1990, "Prefix Sums and Their Applications)

Applications of Scan

- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential :

```
for(j=1;j<n;j++)
   out[j] = out[j-1] + f(j);
```

  - into parallel:

```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

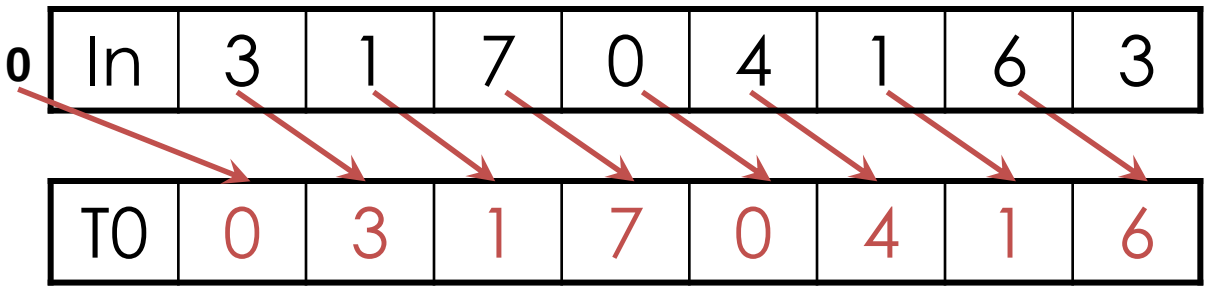
- Useful for many parallel algorithms:
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - **Building data structures**
  - Etc.

Scan on a serial CPU

```
void scan( float* scanned, float* input, int length)
{
  scanned[0] = 0;
  for(int i = 1; i < length; ++i)
  {
    scanned[i] = input[i-1] + scanned[i-1];
  }
}
```

- Just add each element to the sum of the elements before it
- Trivial, but sequential
- Exactly ***n*** adds: optimal

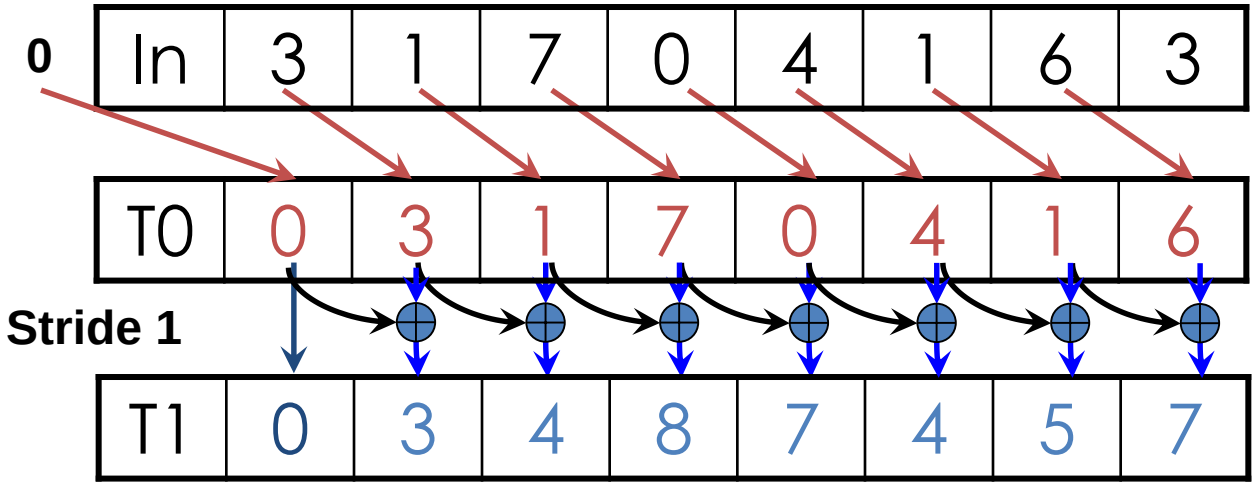
A First-Attempt Parallel Scan Algorithm



Each UE reads one value from the input array in device memory into shared memory array T0. UE 0 writes 0 into shared memory array.

1. Read input to shared memory. Set first element to zero and shift others right by one.

A First-Attempt Parallel Scan Algorithm

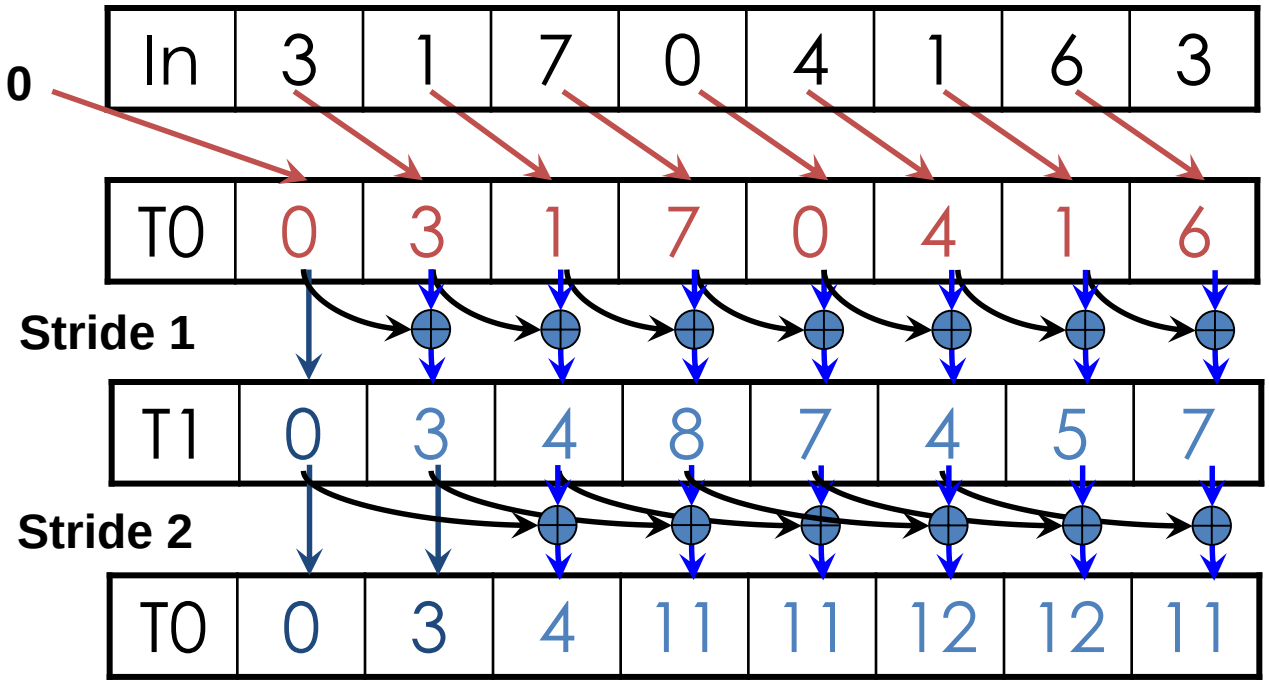


1. (previous slide)
2. Iterate log(n) times: UEs *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #1
Stride = 1

- Active UEs: *stride* to *n*-1 (*n-stride* UEs)
- UE *j* adds elements *j* and *j-stride* from T0 and writes result into shared memory buffer T1 (ping-pong)

A First-Attempt Parallel Scan Algorithm

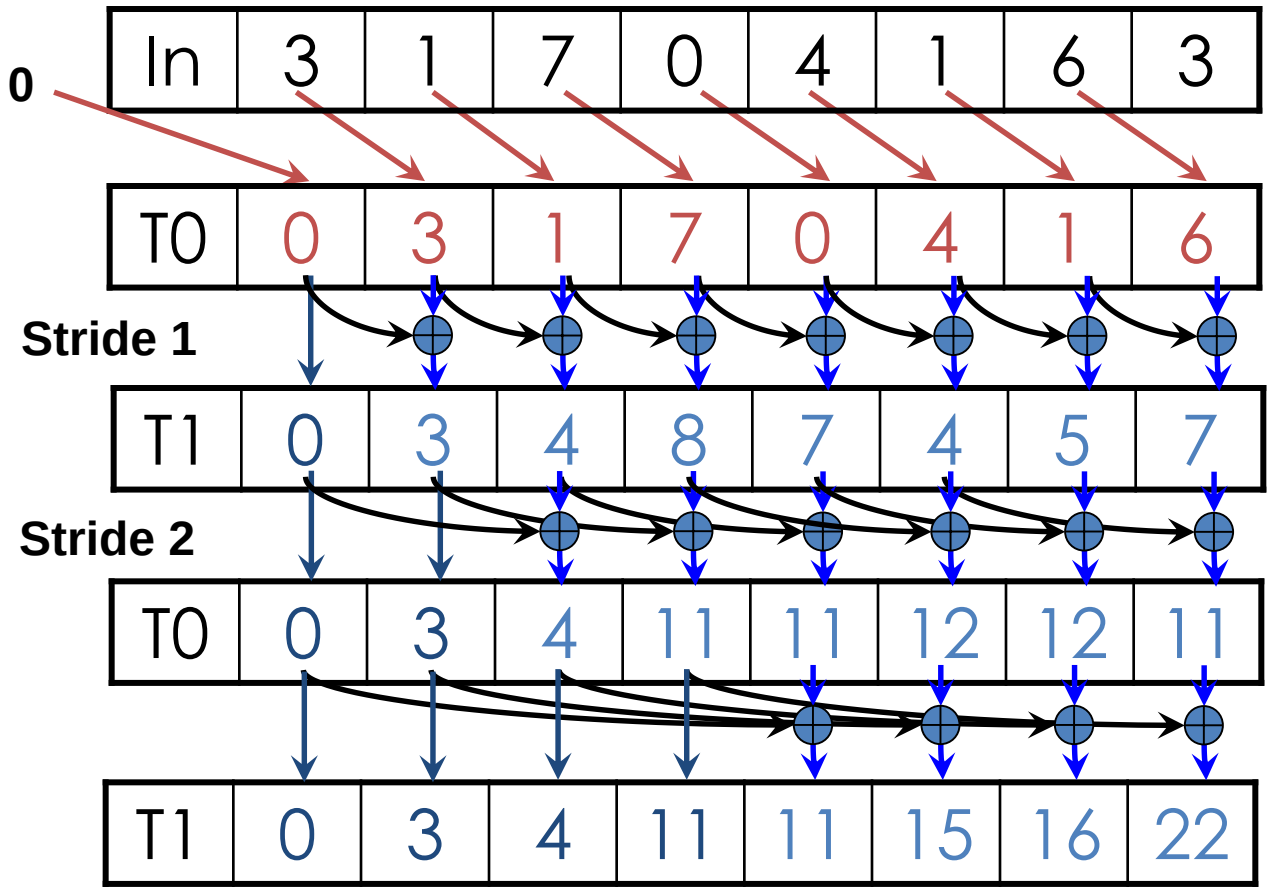


1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: UEs *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #2
Stride = 2

A First-Attempt Parallel Scan Algorithm

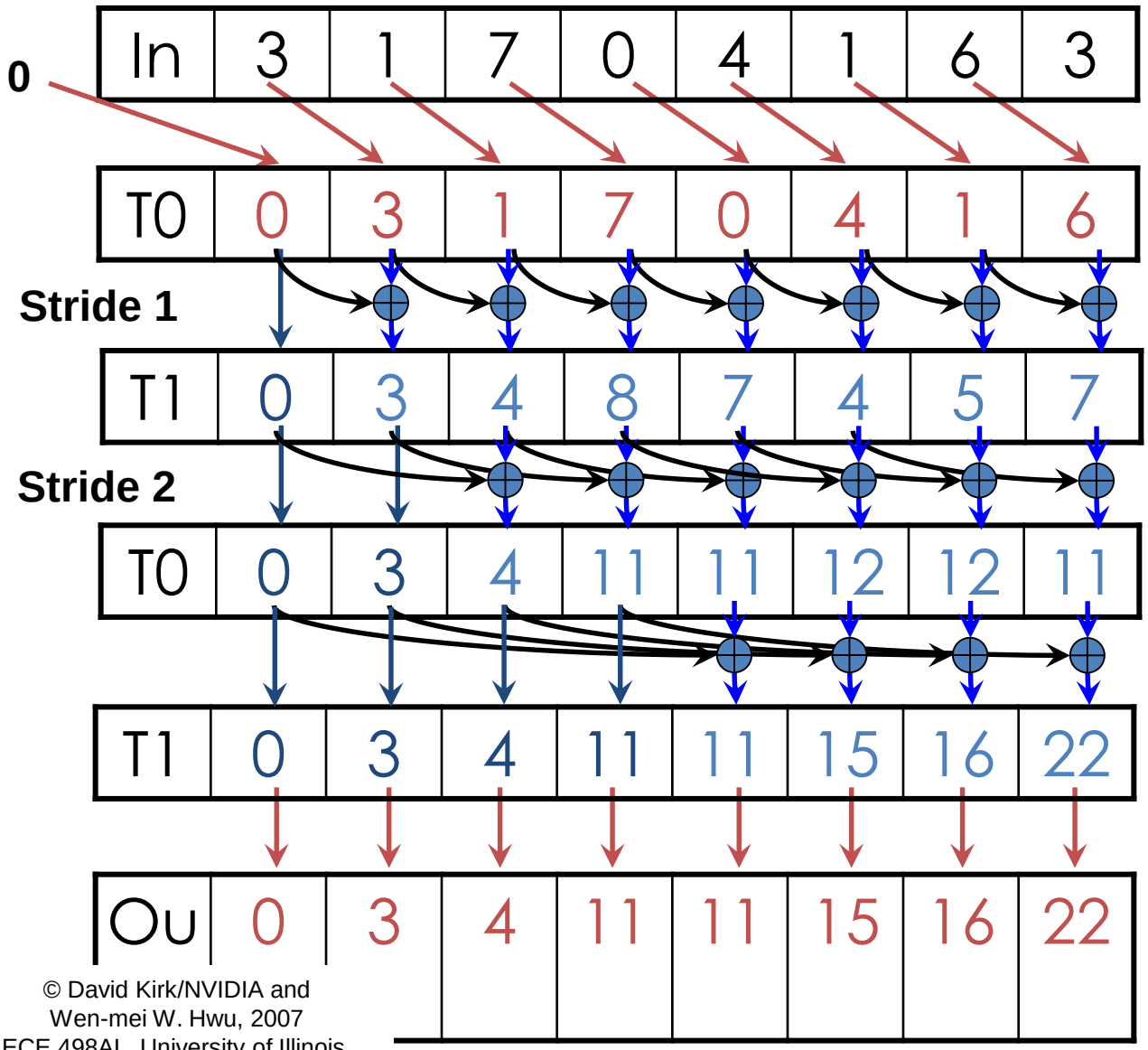


1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

2. Iterate log(n) times: UEs *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

Iteration #3
Stride = 4

A First-Attempt Parallel Scan Algorithm



1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.
2. Iterate log(n) times: UEs *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)
3. Write output.

What is wrong with our first-attempt parallel scan?

- *Work Efficient:*
  - A parallel algorithm is work efficient if it does the same amount of work as an optimal sequential complexity

- Scan executes log(n) parallel iterations
  - The steps do n-1, n-2, n-4,... n/2 adds each
  - Total adds: n * (log(n) – 1) + 1 → O(n*log(n)) work

- This scan algorithm is NOT work efficient
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 10^6 elements!

Improving Efficiency

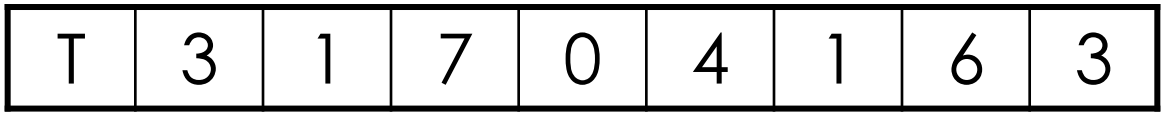
- A common parallel algorithm pattern:

  Balanced Trees

  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each UE does at each step

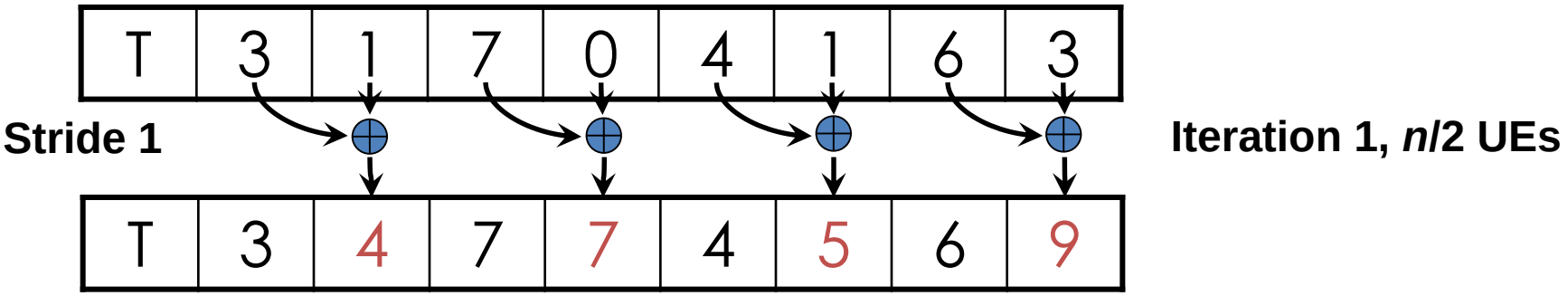
- For scan:
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

Build the Sum Tree



Assume array is already in shared memory

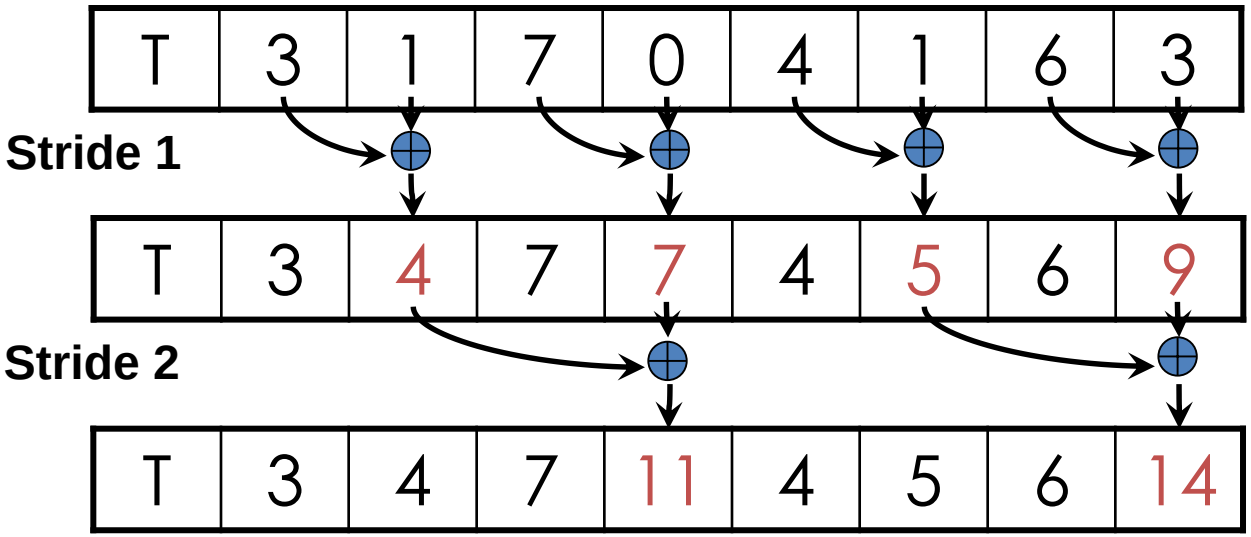
Build the Sum Tree



Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value

Build the Sum Tree

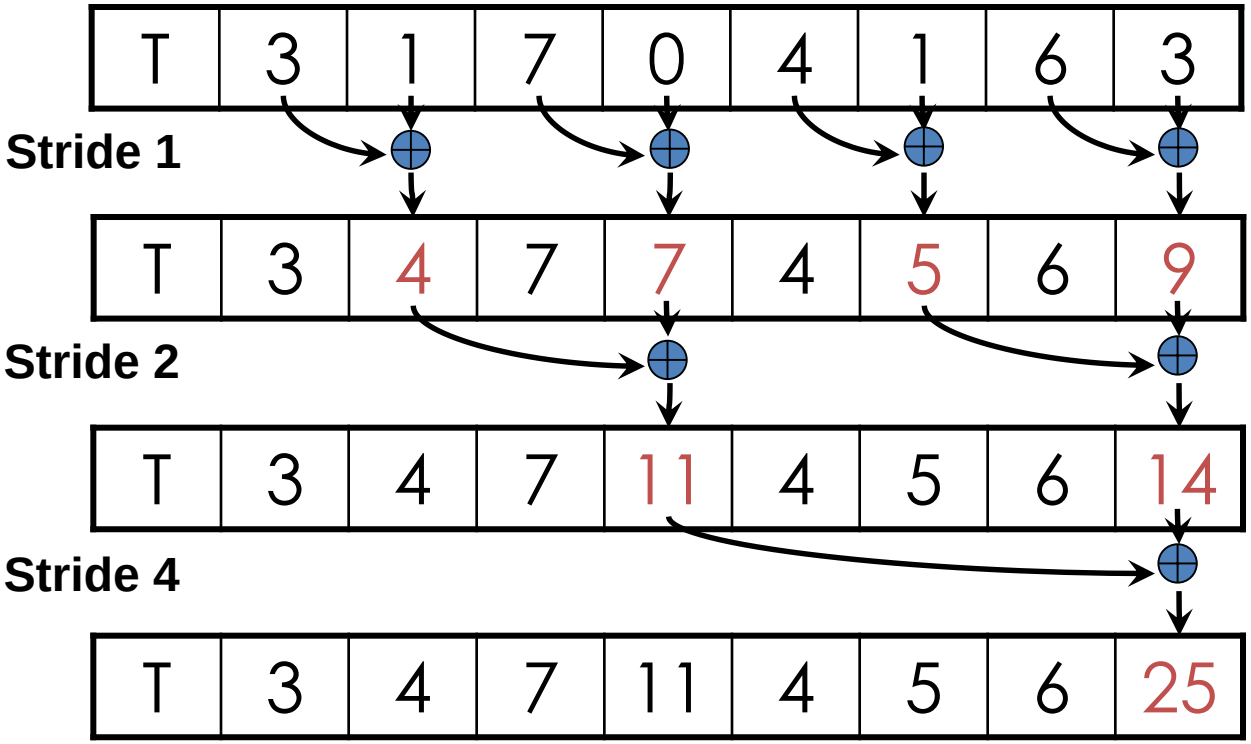


Iteration 2, *n*/4 UEs

Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value

Build the Sum Tree



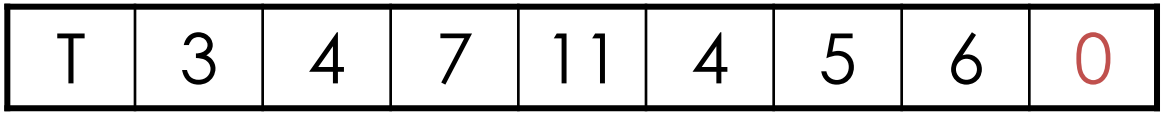
Iteration log(*n*), 1 UE

Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value.

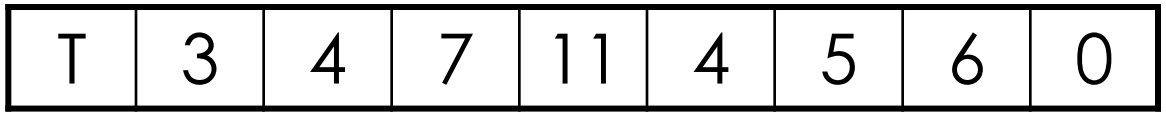
Note that this algorithm operates in-place: no need for double buffering

Zero the Last Element

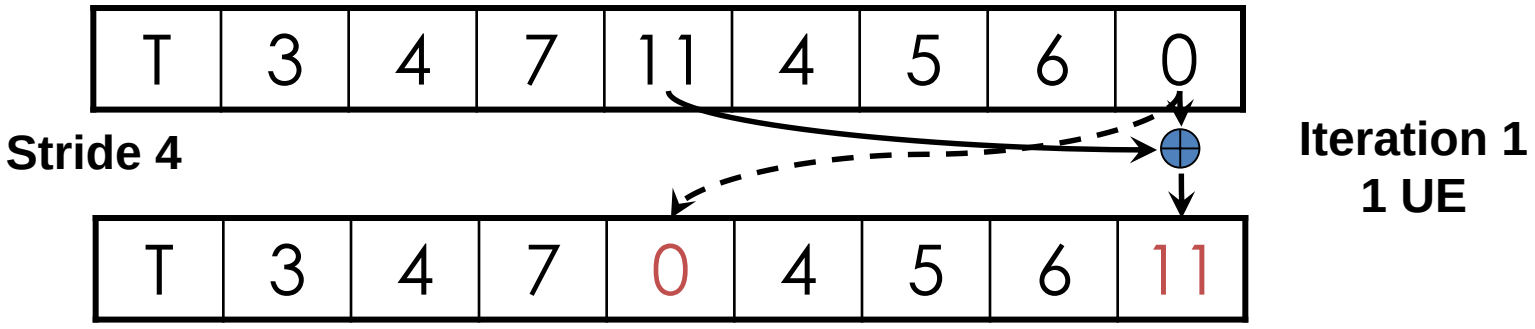


We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

Build Scan From Partial Sums



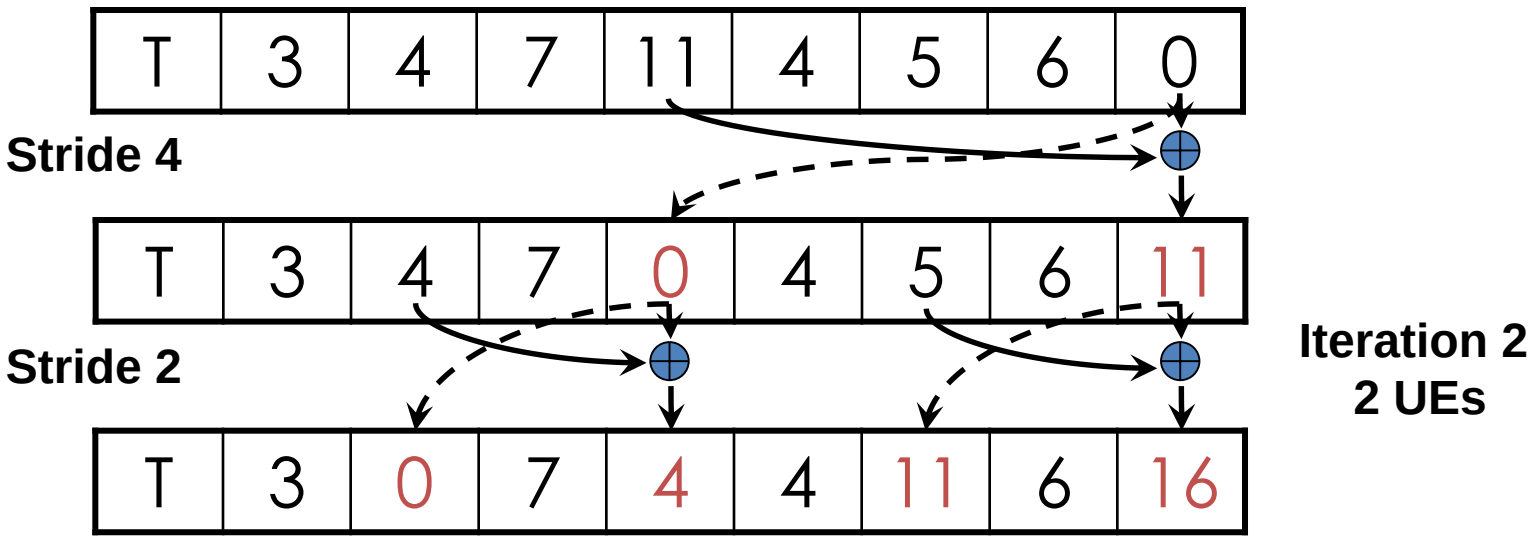
Build Scan From Partial Sums



Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

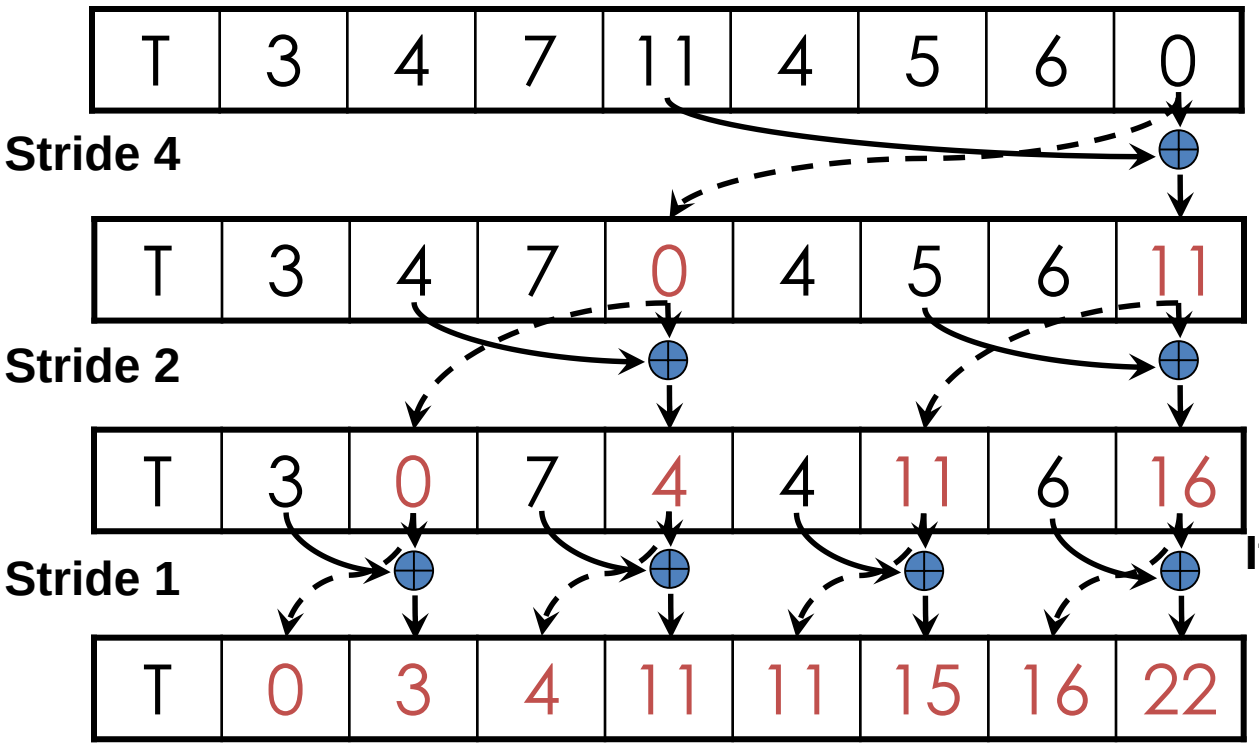
Build Scan From Partial Sums



Each ⊕ corresponds to a single UE.

Iterate log(n) times. Each UE adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Build Scan From Partial Sums



Done! We now have a completed scan that we can write out to device memory.

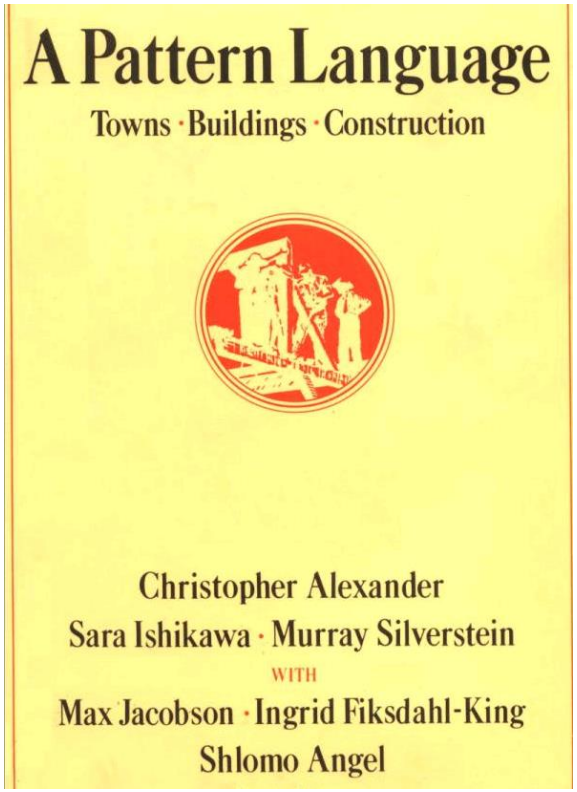
Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

Building Data Structures with Scans

- Fun on the board

History

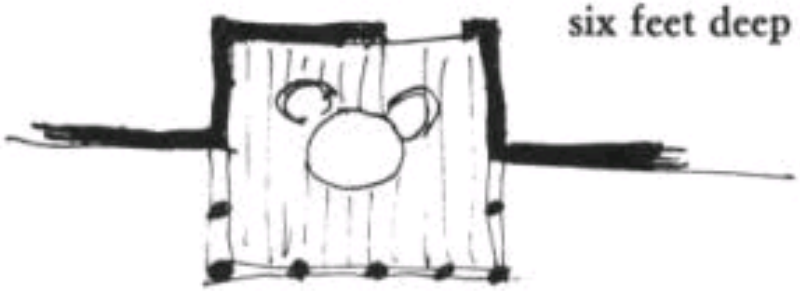
- Berkeley architecture professor Christopher Alexander
- In 1977, patterns for city planning, landscaping, and architecture in an attempt to capture principles for "living" design



Example 167 (p. 783): 6ft Balcony

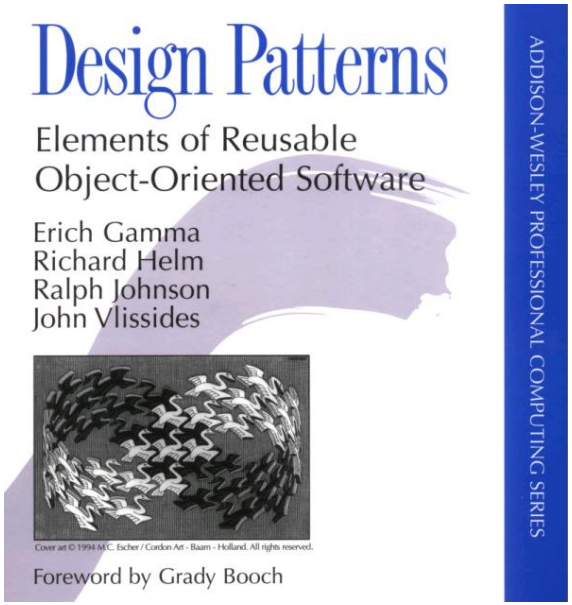
Therefore:

Whenever you build a balcony, a porch, a gallery, or a terrace always make it at least six feet deep. If possible, recess at least a part of it into the building so that it is not cantilevered out and separated from the building by a simple line, and enclose it partially.



Patterns in Object-Oriented Programming

- Design Patterns: Elements of Reusable Object-Oriented Software (1995)
  - Gang of Four (GOF): Gamma, Helm, Johnson, Vlissides
  - Catalogue of patterns
  - Creation, structural, behavioral



Patterns for Parallelizing Programs

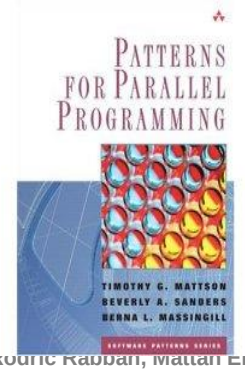
4 Design Spaces

Algorithm Expression

- Finding Concurrency
  - Expose concurrent tasks
- Algorithm Structure
  - Map tasks to processes to exploit parallel architecture

Software Construction

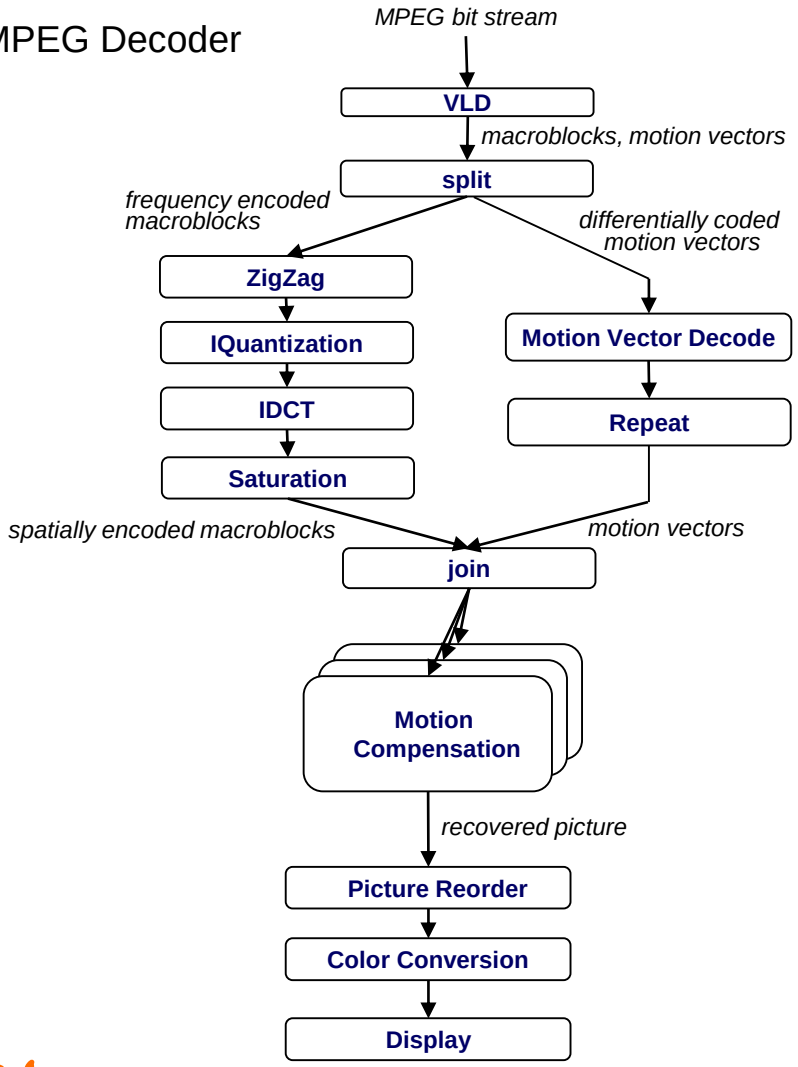
- Supporting Structures
  - Code and data structuring patterns
- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs



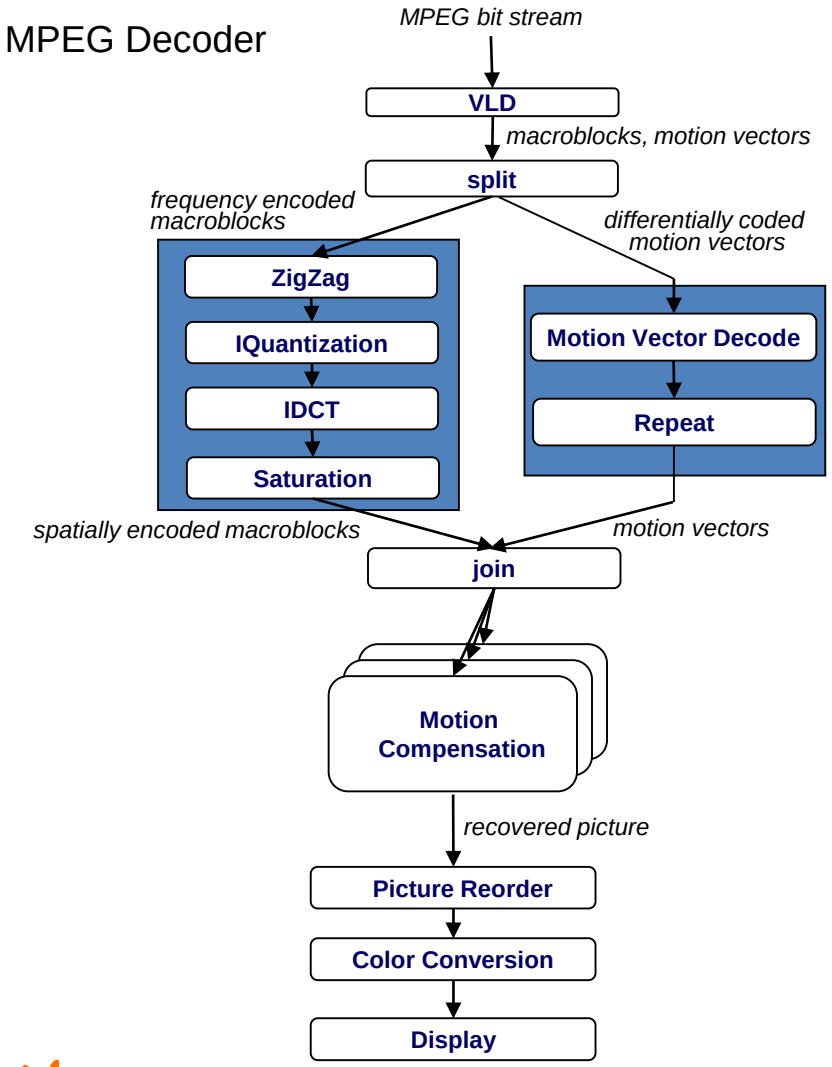
Patterns for Parallel Programming. Mattson, Sanders, and Massingill (2005).

Here's my algorithm. Where's the concurrency?

MPEG Decoder

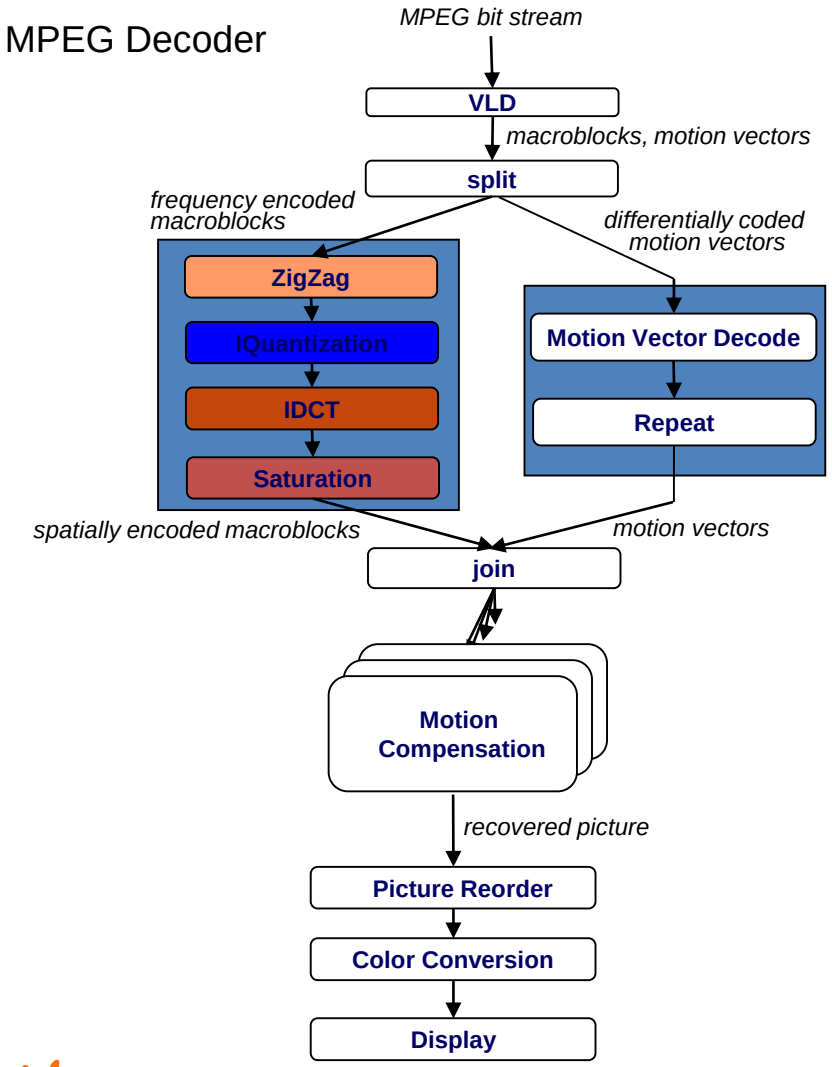


Here's my algorithm. Where's the concurrency?

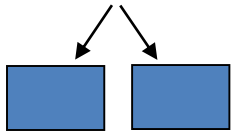


- Task decomposition
  - Independent coarse-grained computation
  - Inherent to algorithm
- Sequence of statements (instructions) that operate together as a group
  - Corresponds to some logical part of program
  - Usually follows from the way programmer thinks about a problem

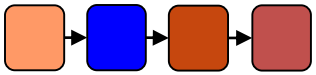
Here's my algorithm. Where's the concurrency?



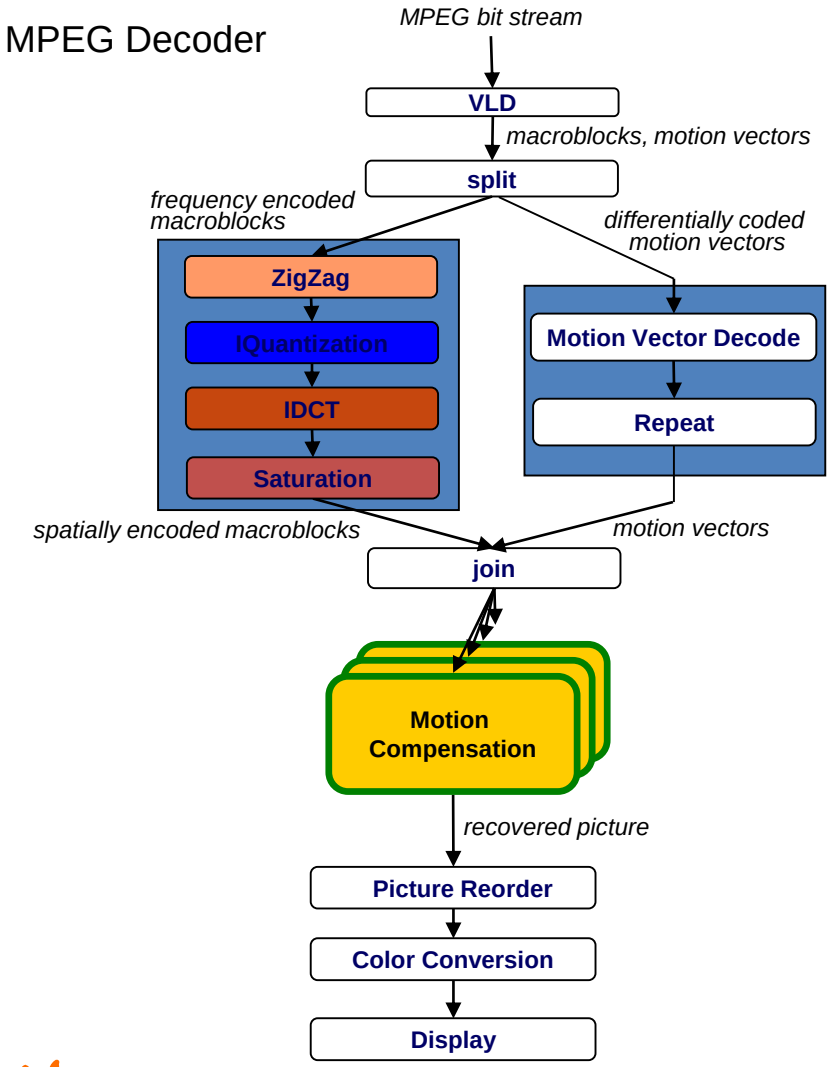
- Task decomposition
  - Parallelism in the application



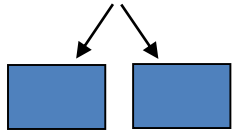
- Pipeline task decomposition
  - Data assembly lines
  - Producer-consumer chains



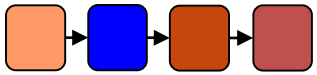
Here's my algorithm. Where's the concurrency?



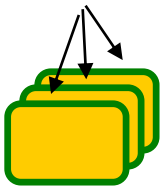
- Task decomposition
  - Parallelism in the application



- Pipeline task decomposition
  - Data assembly lines
  - Producer-consumer chains



- Data decomposition
  - Same computation is applied to small data chunks derived from large data set



Guidelines for Task Decomposition

- Algorithms start with a good understanding of the problem being solved

- Programs often naturally decompose into tasks
  - Two common decompositions are
    - Function calls and
    - Distinct loop iterations

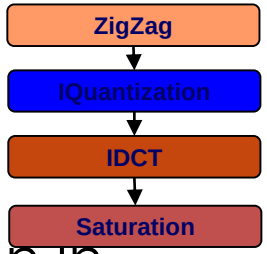
- Easier to start with many tasks and later fuse them, rather than too few tasks and later try to split them

Guidelines for Task Decomposition

- Flexibility
  - Program design should afford flexibility in the number and size of tasks generated
    - Tasks should not tied to a specific architecture
    - Fixed tasks vs. Parameterized tasks
- Efficiency
  - Tasks should have enough work to amortize the cost of creating and managing them
  - Tasks should be sufficiently independent so that managing dependencies doesn't become the bottleneck
- Simplicity
  - The code has to remain readable and easy to understand, and debug

Case for Pipeline Decomposition

- Data is flowing through a sequence of stages
  - Assembly line is a good analogy



- What's a prime example of pipeline decomposition in computer architecture?
  - Instruction pipeline in modern CPUs

- What's an example pipeline you may use in your UNIX shell?
  - Pipes in UNIX: cat foobar.c | grep bar | wc

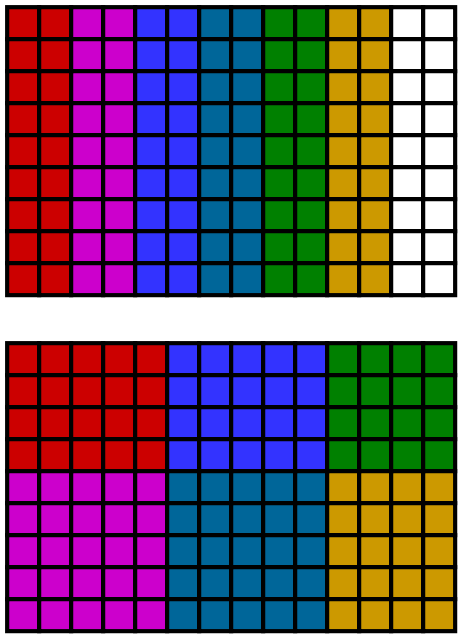
- Other examples
  - Signal processing
  - Graphics

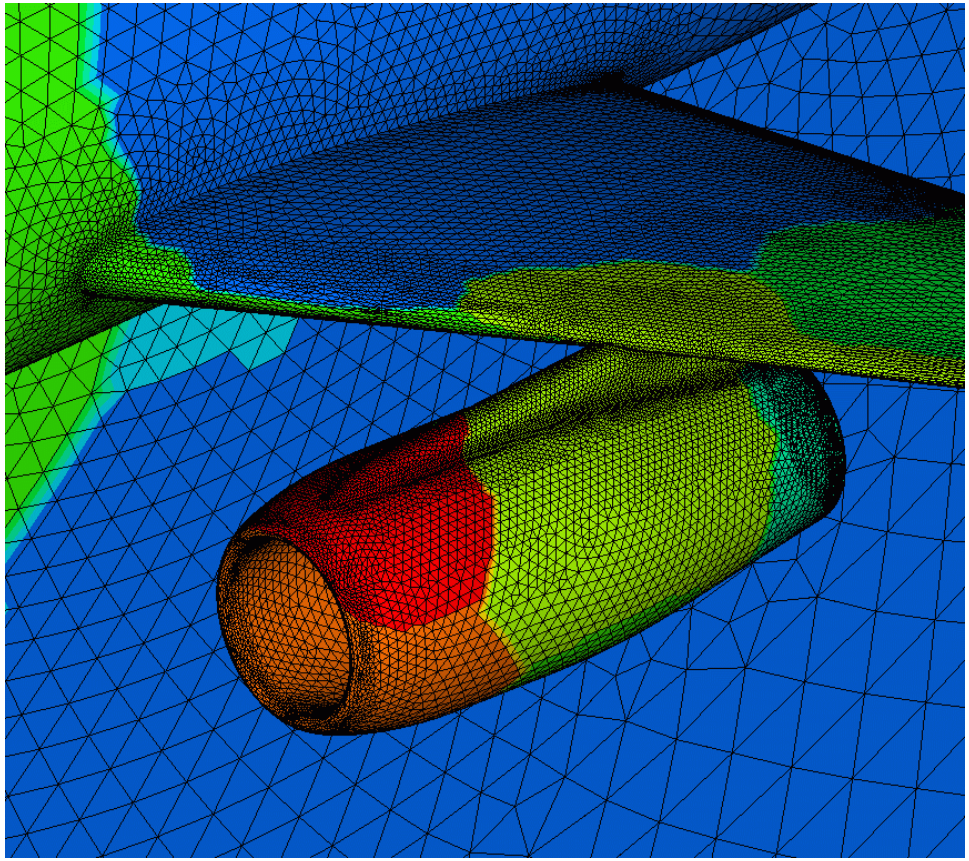
Guidelines for Data Decomposition

- Data decomposition is often implied by task decomposition
- Programmers need to address task and data decomposition to create a parallel program
  - Which decomposition to start with?
- Data decomposition is a good starting point when
  - Main computation is organized around manipulation of a large data structure
  - Similar operations are applied to different parts of the data structure

Common Data Decompositions

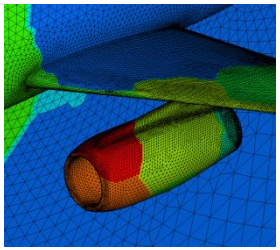
- Geometric data structures
  - Decomposition of arrays along rows, columns, blocks
  - Decomposition of meshes into domains



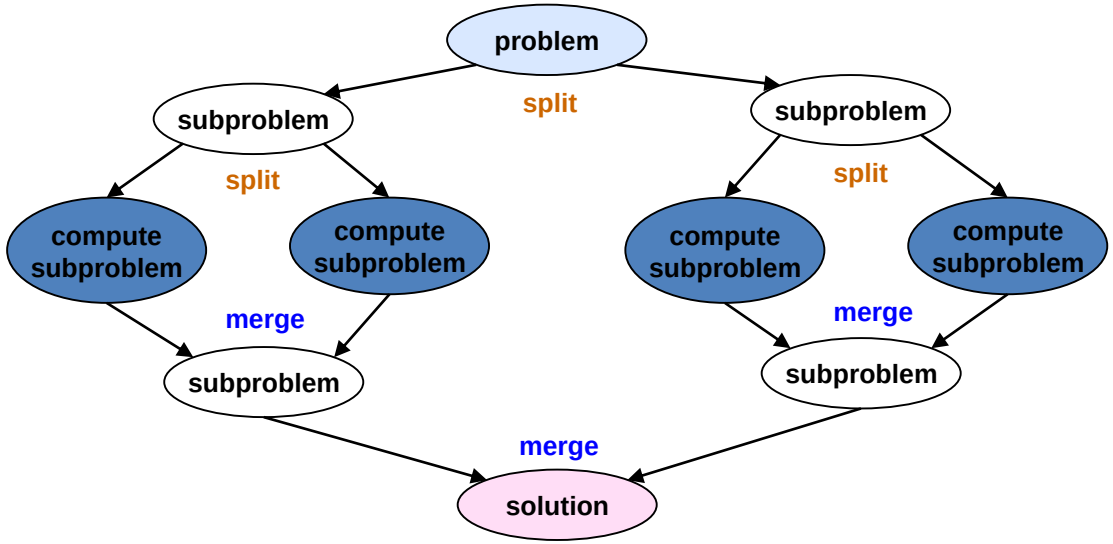


Common Data Decompositions

- Geometric data structures
  - Decomposition of arrays along rows, columns, blocks
  - Decomposition of meshes into domains



- Recursive data structures
  - Example: decomposition of trees into sub-trees

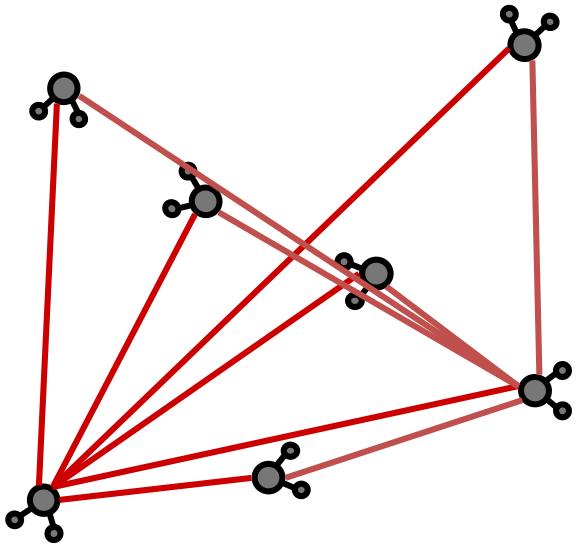


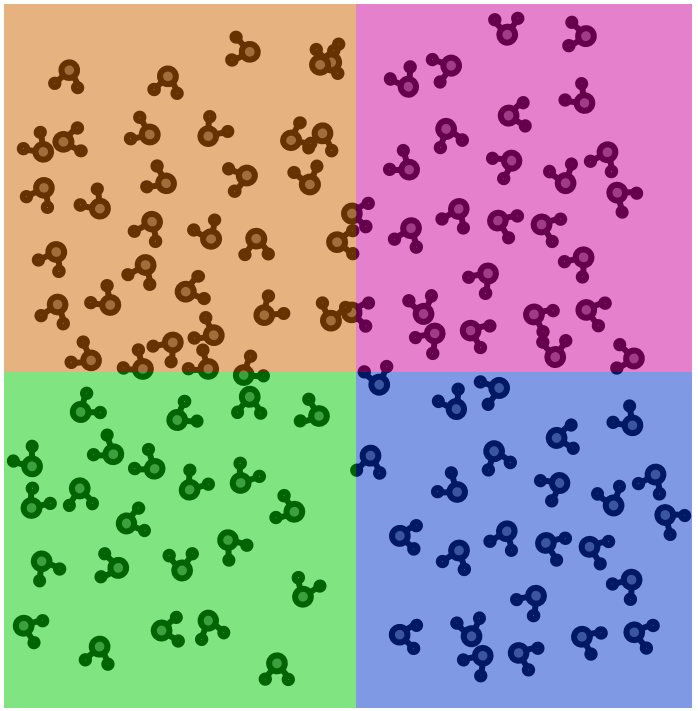
Guidelines for Data Decomposition

- Flexibility
  - Size and number of data chunks should support a wide range of executions
- Efficiency
  - Data chunks should generate comparable amounts of work (for load balancing)
- Simplicity
  - Complex data compositions can get difficult to manage and debug

Data Decomposition Examples

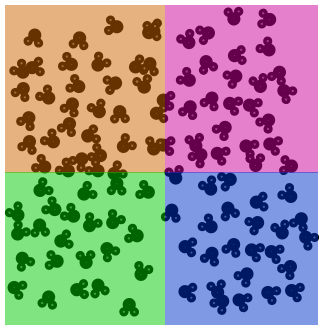
- Molecular dynamics
  - Compute forces
  - Update accelerations and velocities
  - Update positions
- Decomposition
  - Baseline algorithm is N^2
    - All-to-all communication
  - Best decomposition is to treat mols. as a set
  - Some advantages to geometric discussed in future lecture





Data Decomposition Examples

- Molecular dynamics
  - Geometric decomposition



- Merge sort
  - Recursive decomposition