EE382N (20): Computer Architecture - Parallelism and Locality

Lecture 11 – Parallelism in Software II

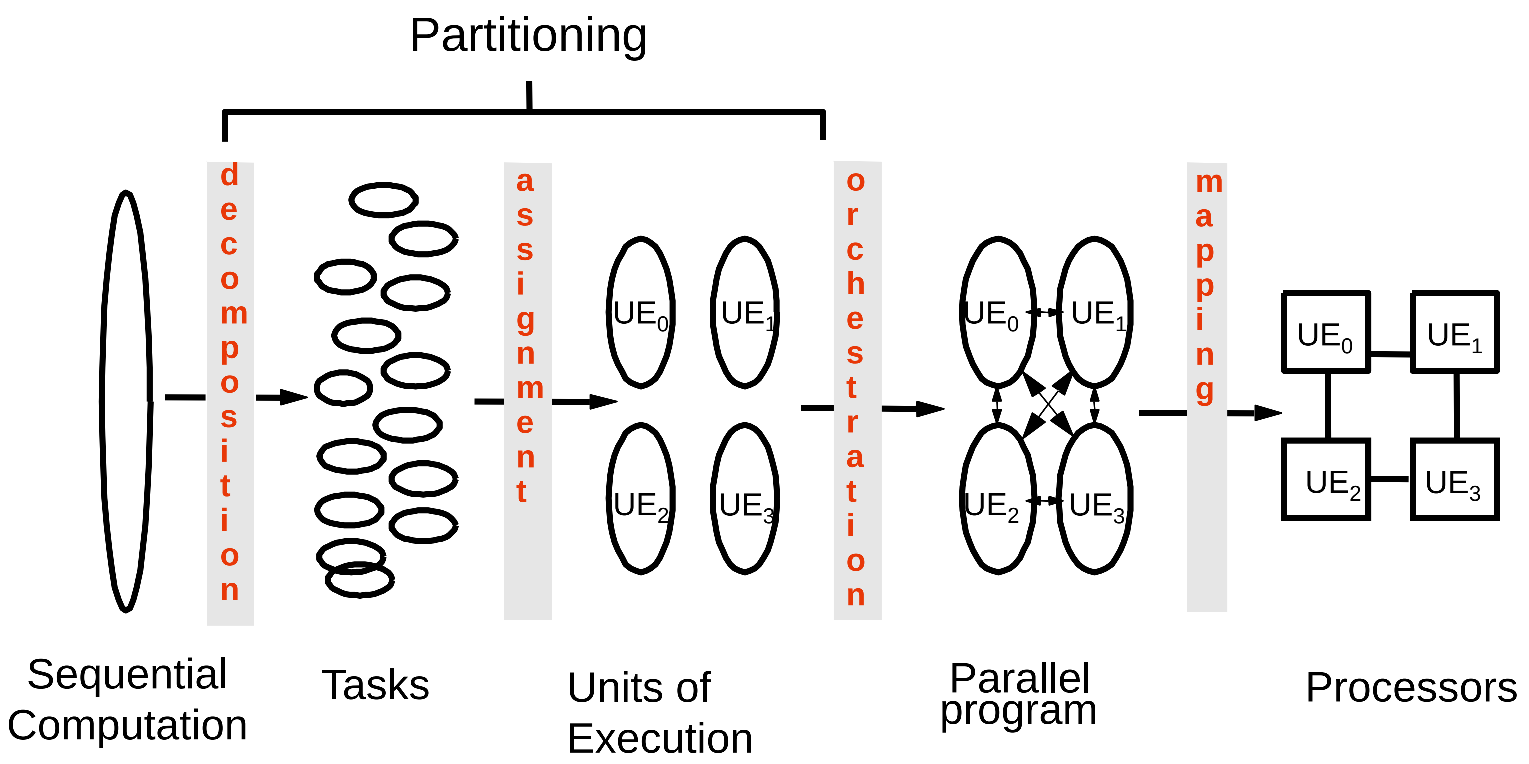
Mattan Erez

The University of Texas at Austin

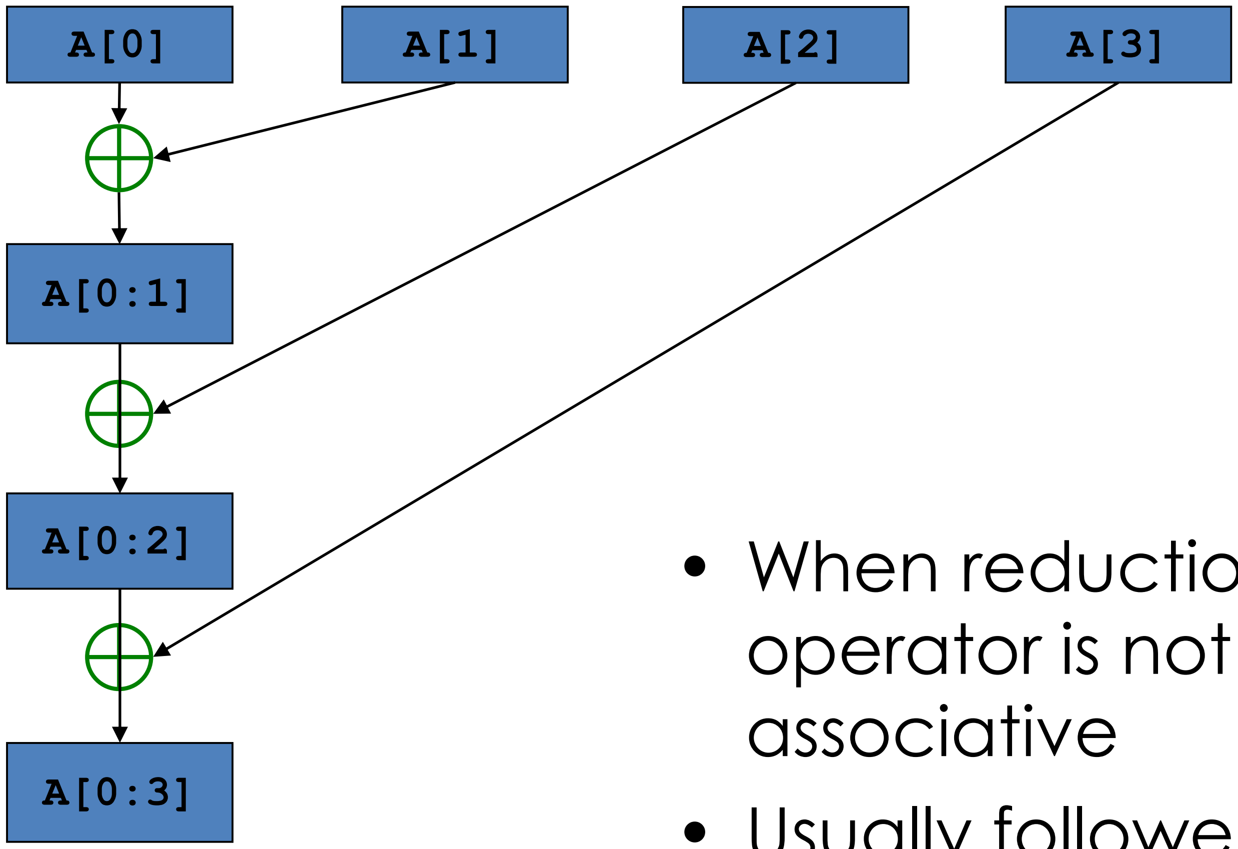
Credits

- 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

4 Common Steps to Creating a Parallel Program

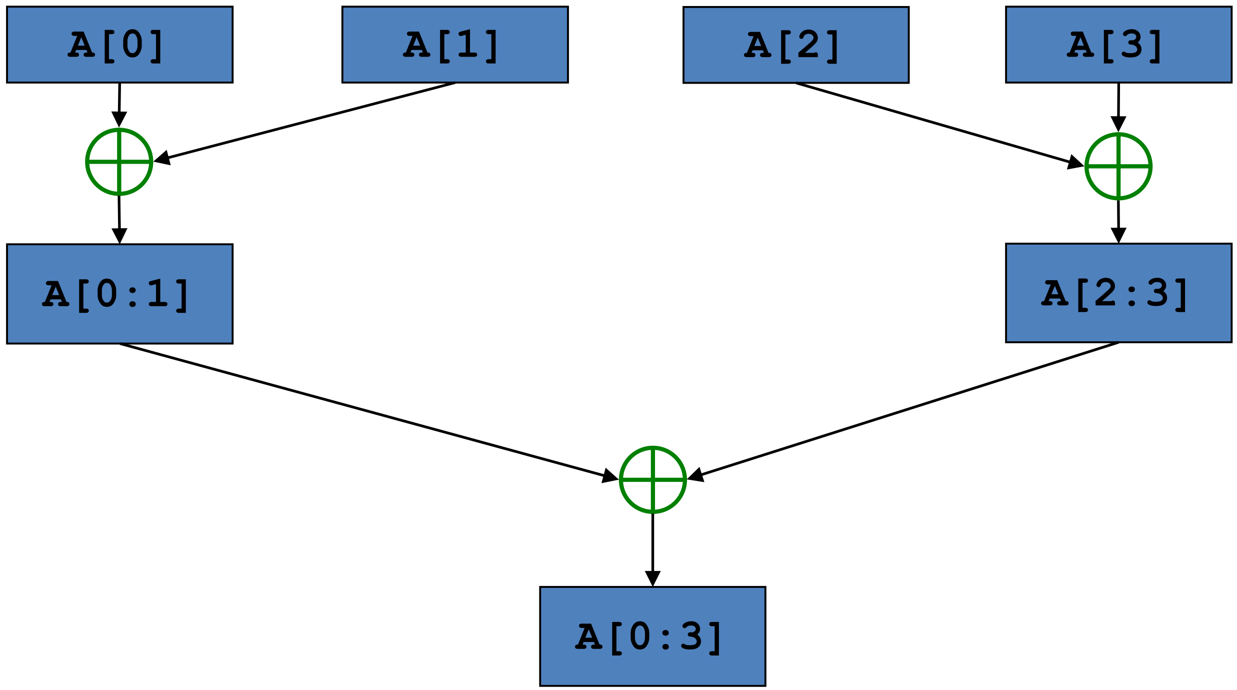


Serial Reduction



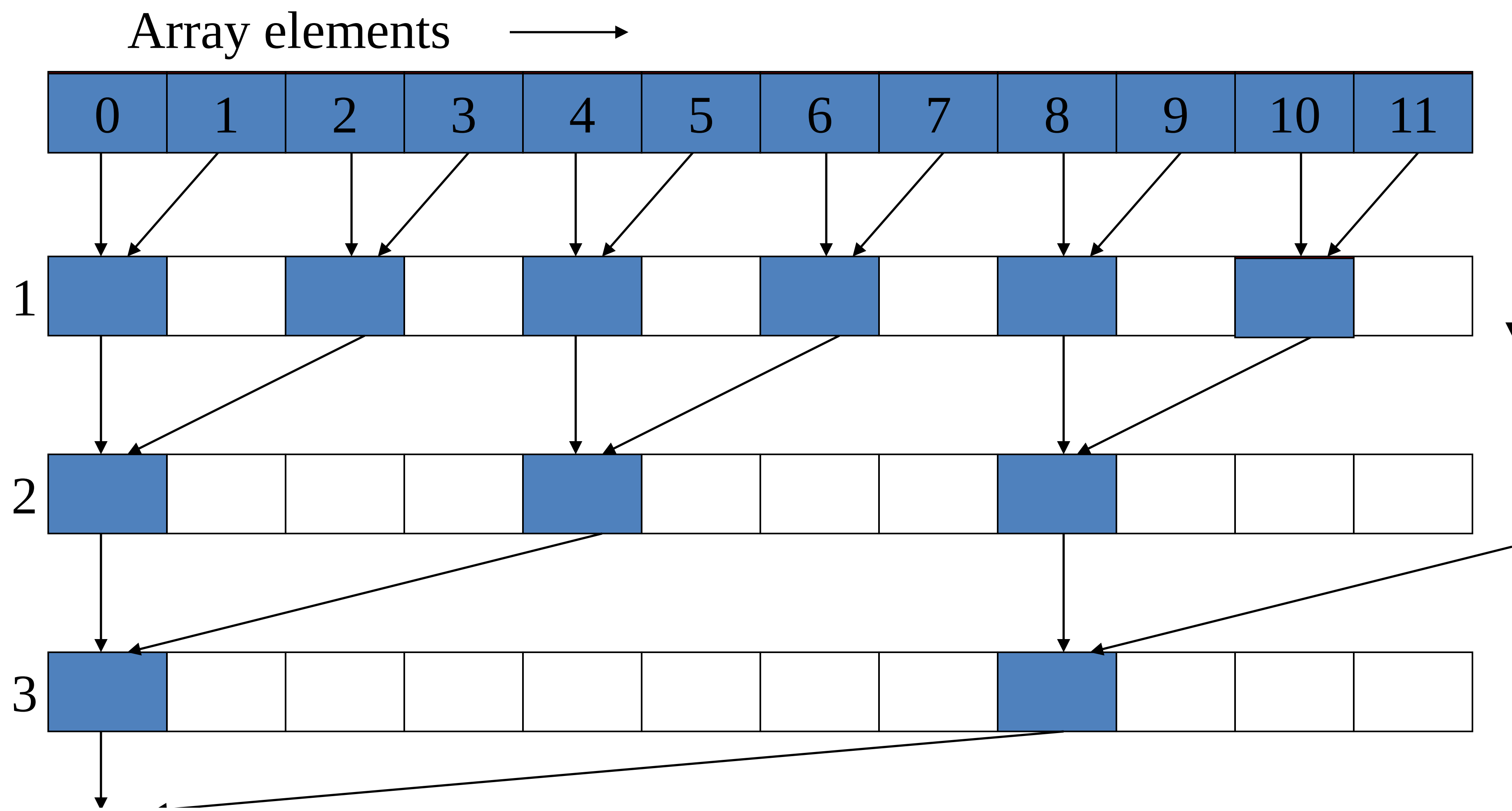
- When reduction operator is not associative
- Usually followed by a broadcast of result

Tree-based Reduction

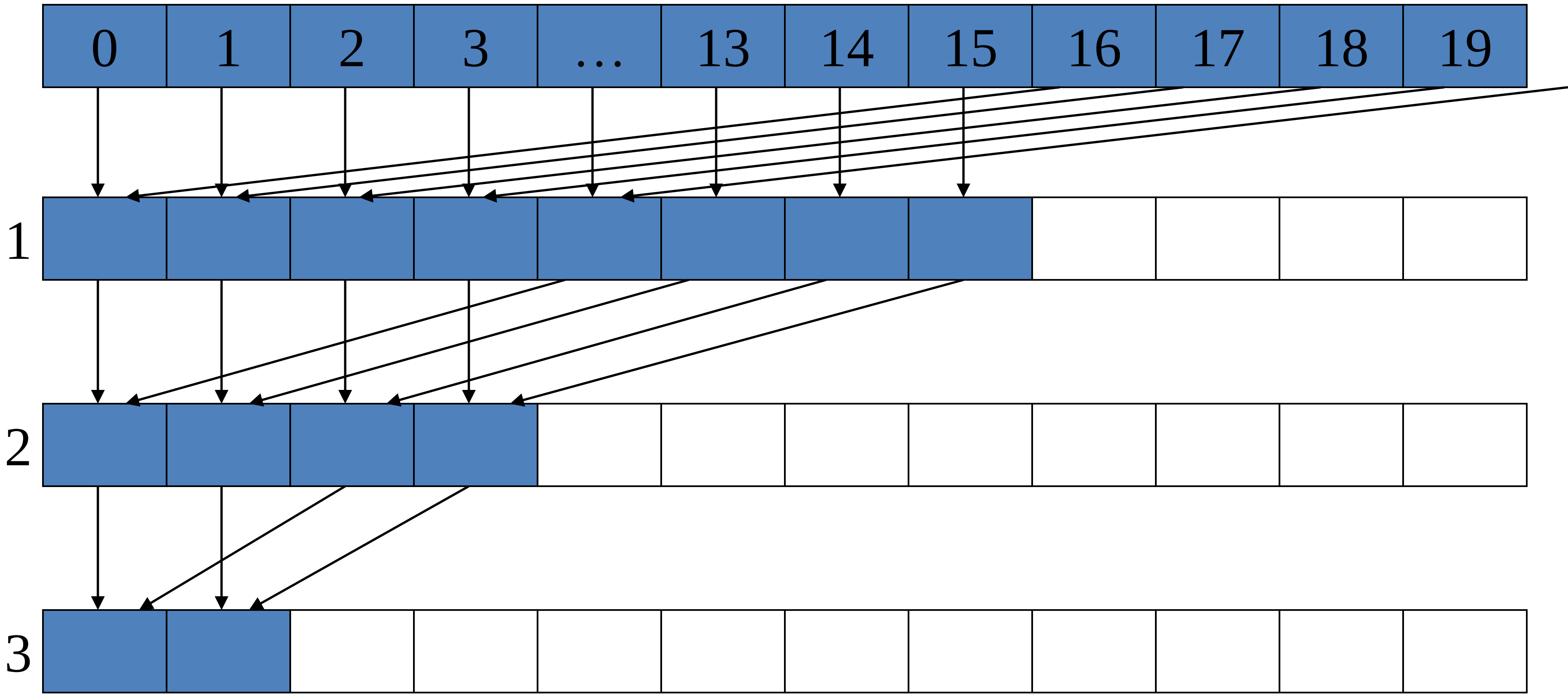


- n steps for 2^n units of execution
- When reduction operator is associative
- Especially attractive when only one task needs result

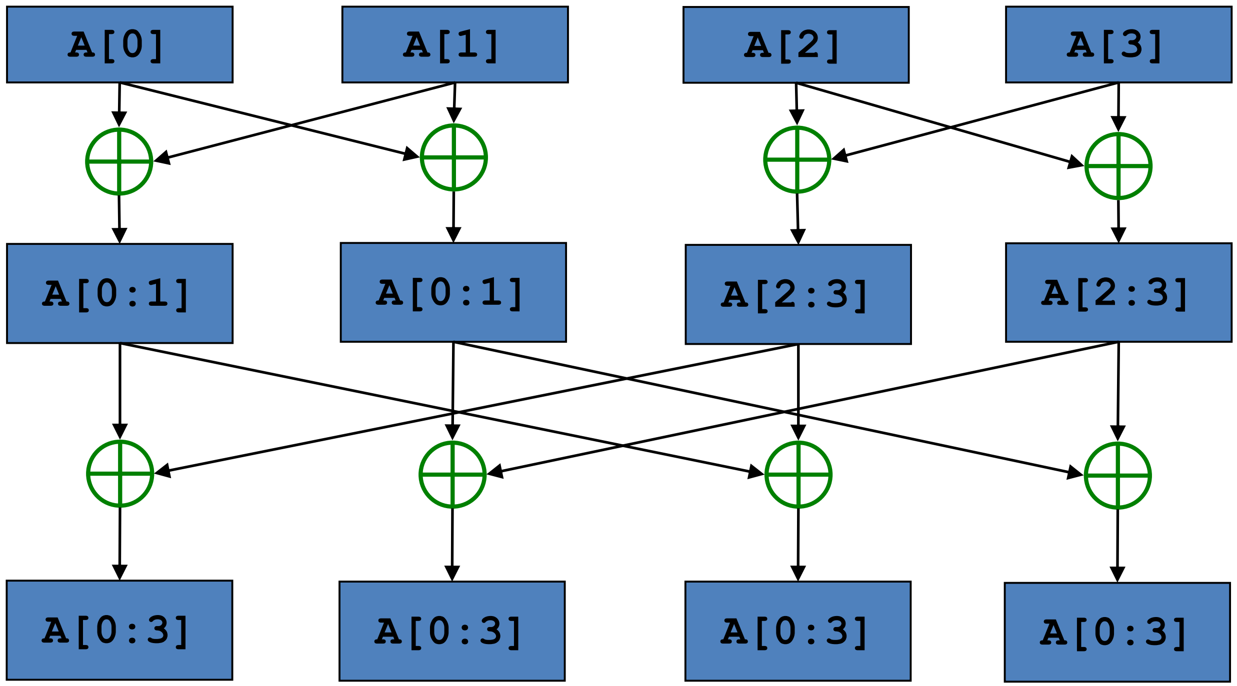
Vector Reduction with Bank Conflicts



No Bank Conflicts



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 set

  [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);
```

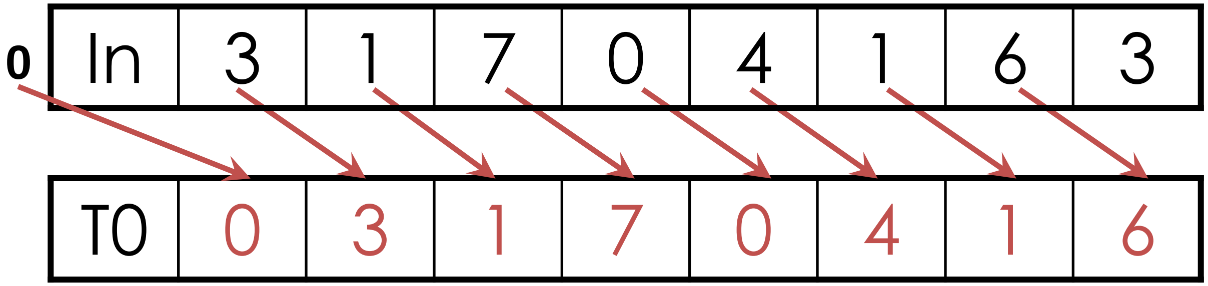
- Useful for many parallel algorithms:
  - radix sort
  - quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - **Building data structures**
  - Etc.

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

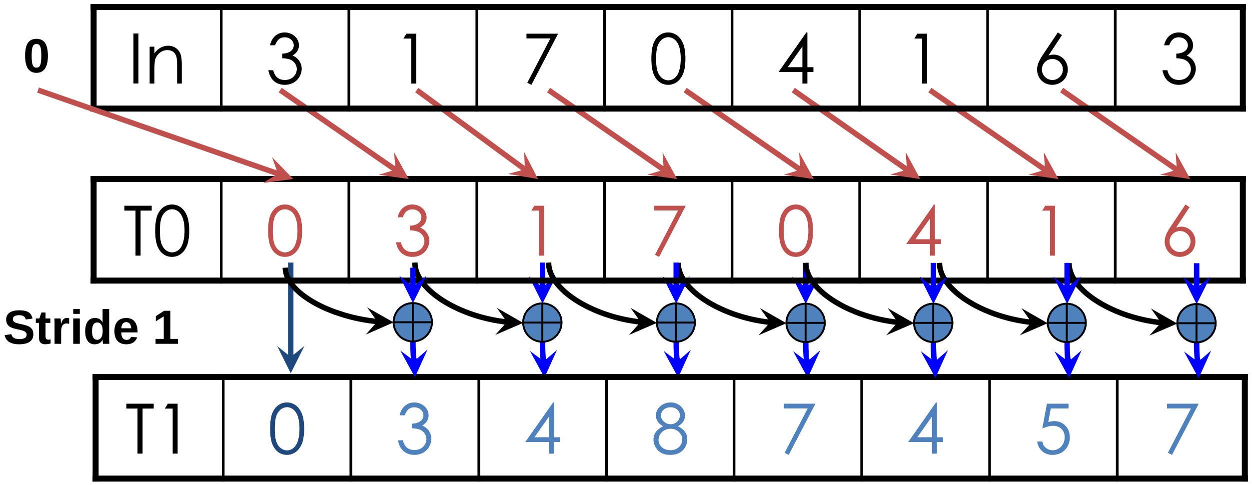
A First-Attempt Parallel Scan Algorithm



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

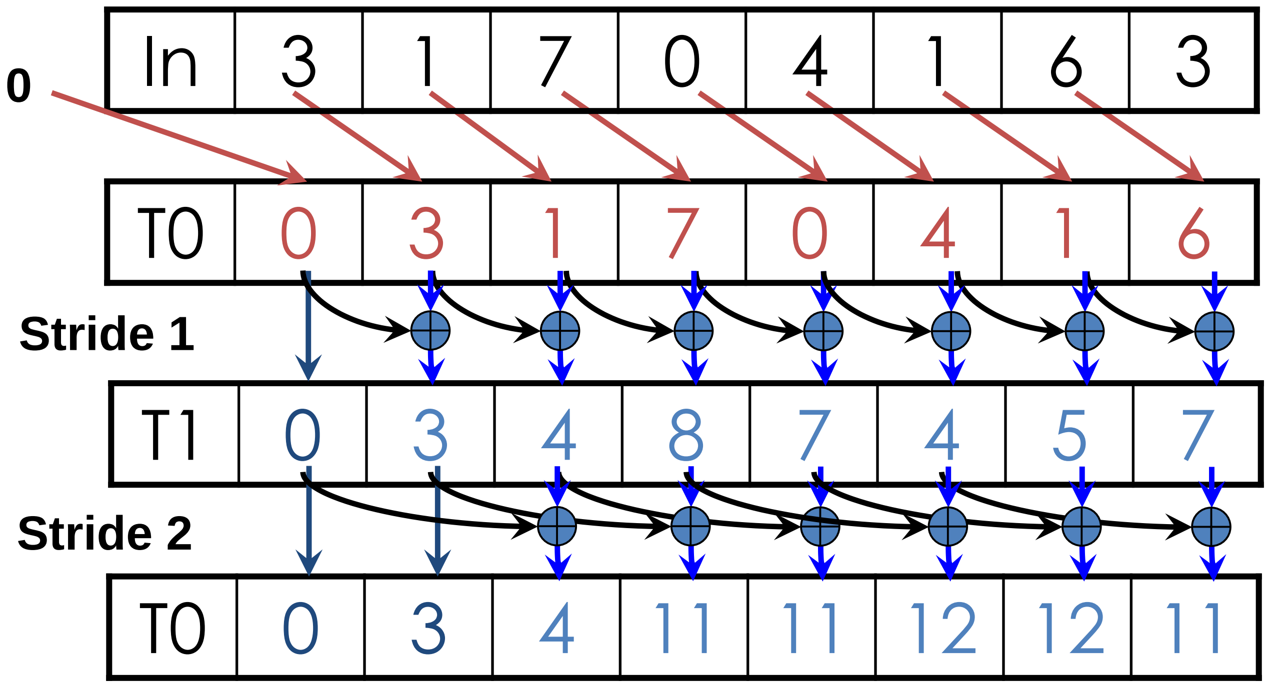


1. (previous slide)
2. Iterate log(n) times: UEs *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (note must double buffer shared mem arrays)

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

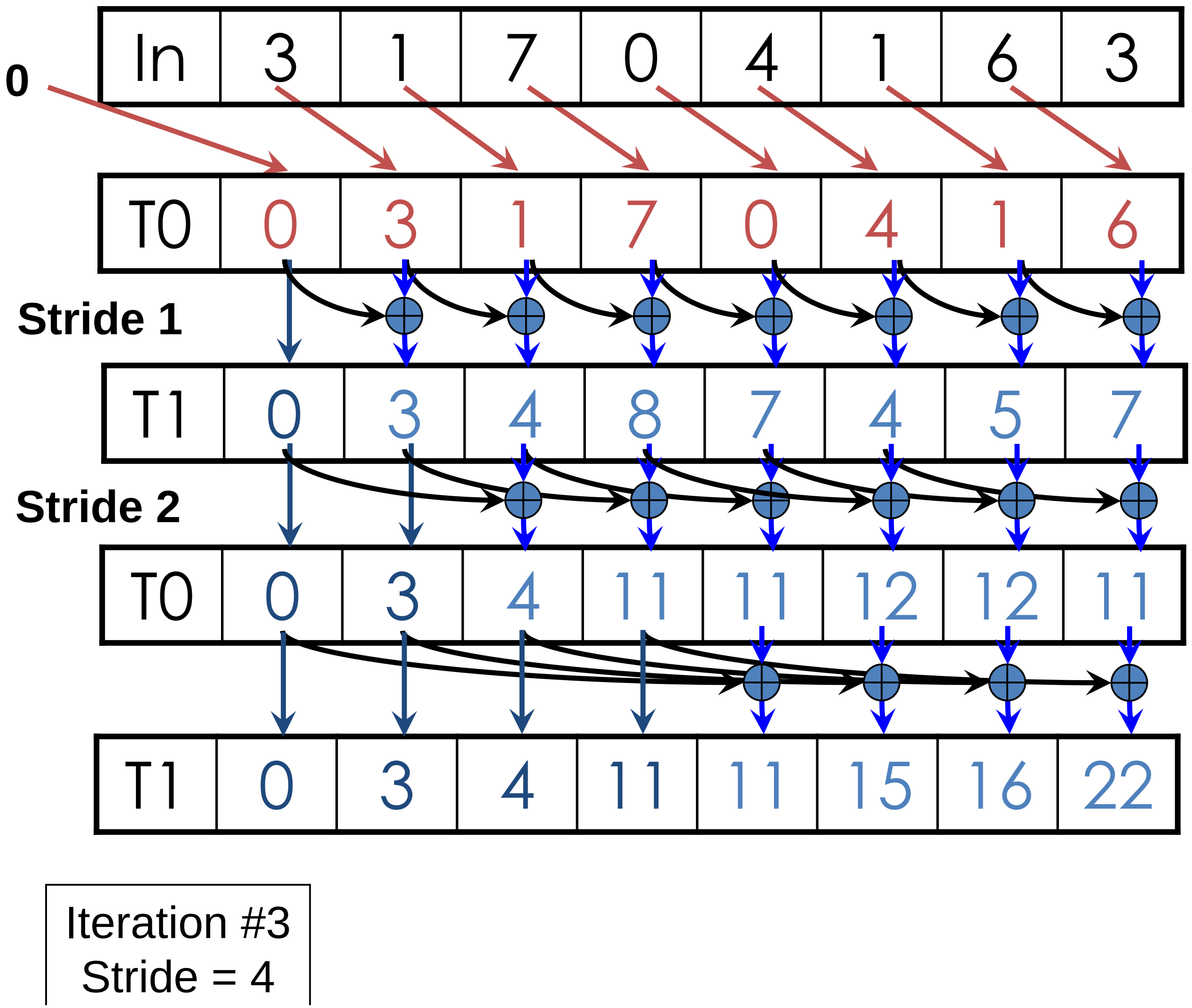


1. Read input from device memory to shared memory. Set first element to zero and shift others right by one.

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)

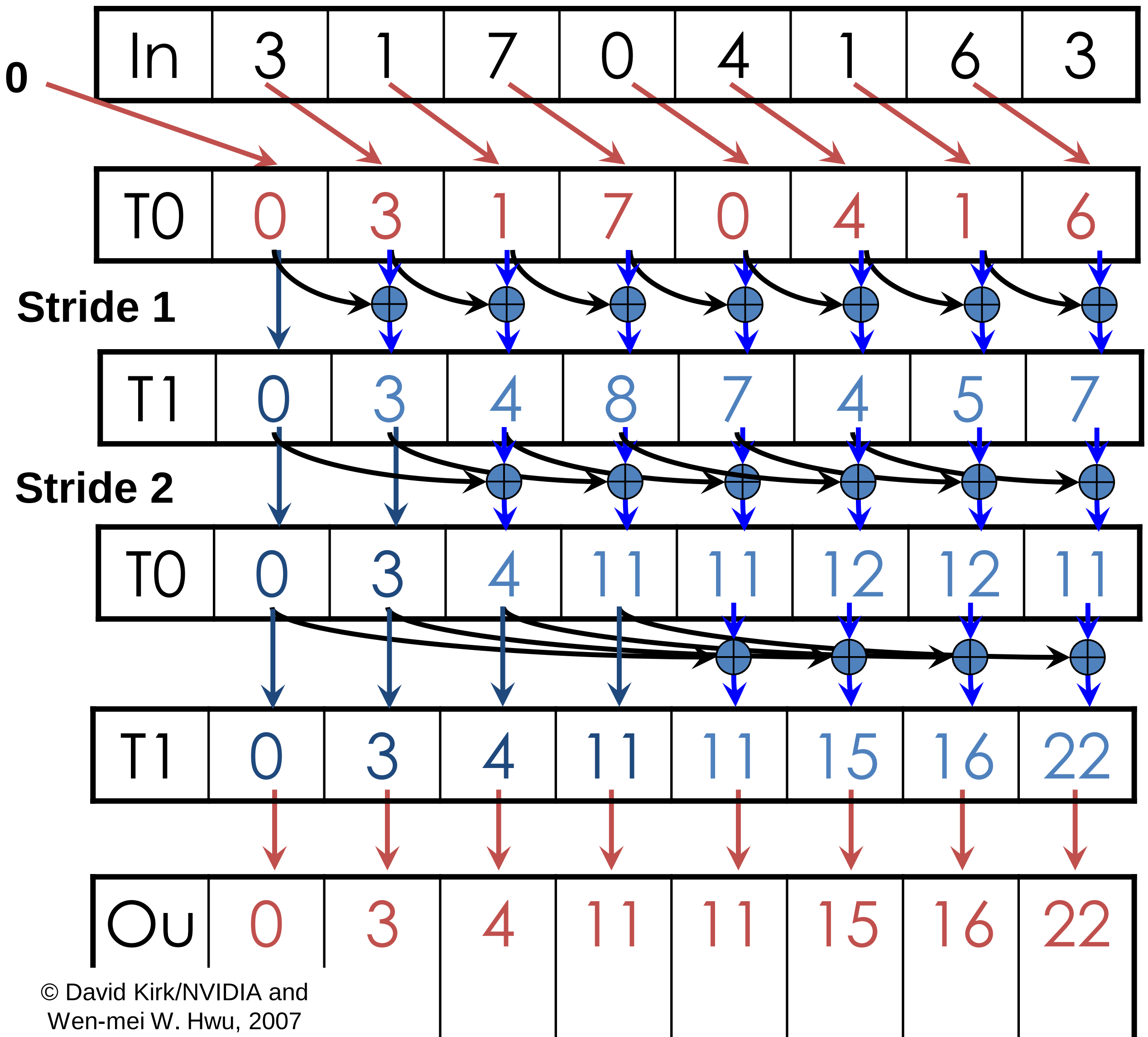
Iteration #2
Stride = 2

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)

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

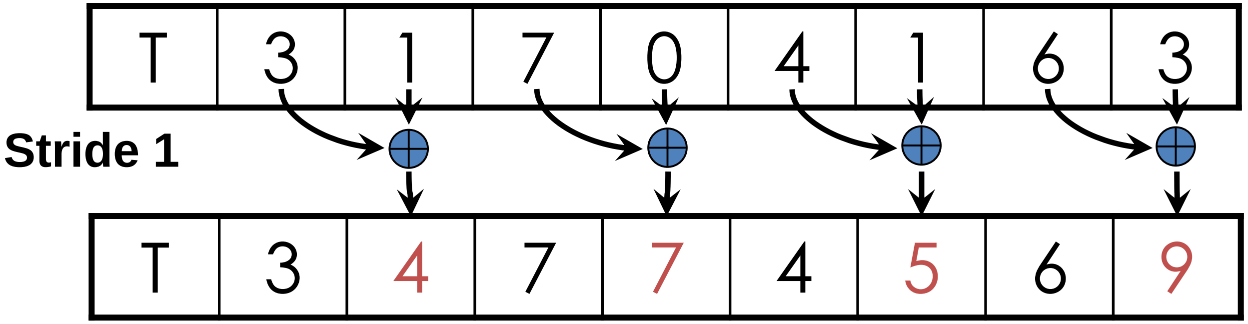
  – Traverse back up the tree building the scan from the partial sums

Build the Sum Tree

T	3	1	7	0	4	1	6	3

Assume array is already in shared memory

Build the Sum Tree

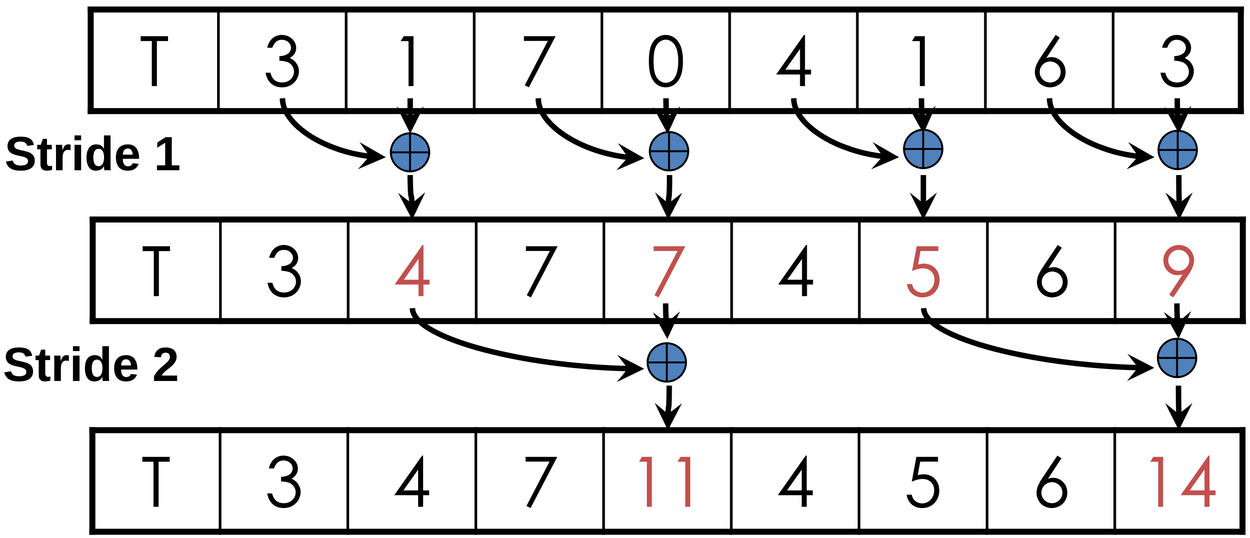


Iteration 1, *n*/2 UEs

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

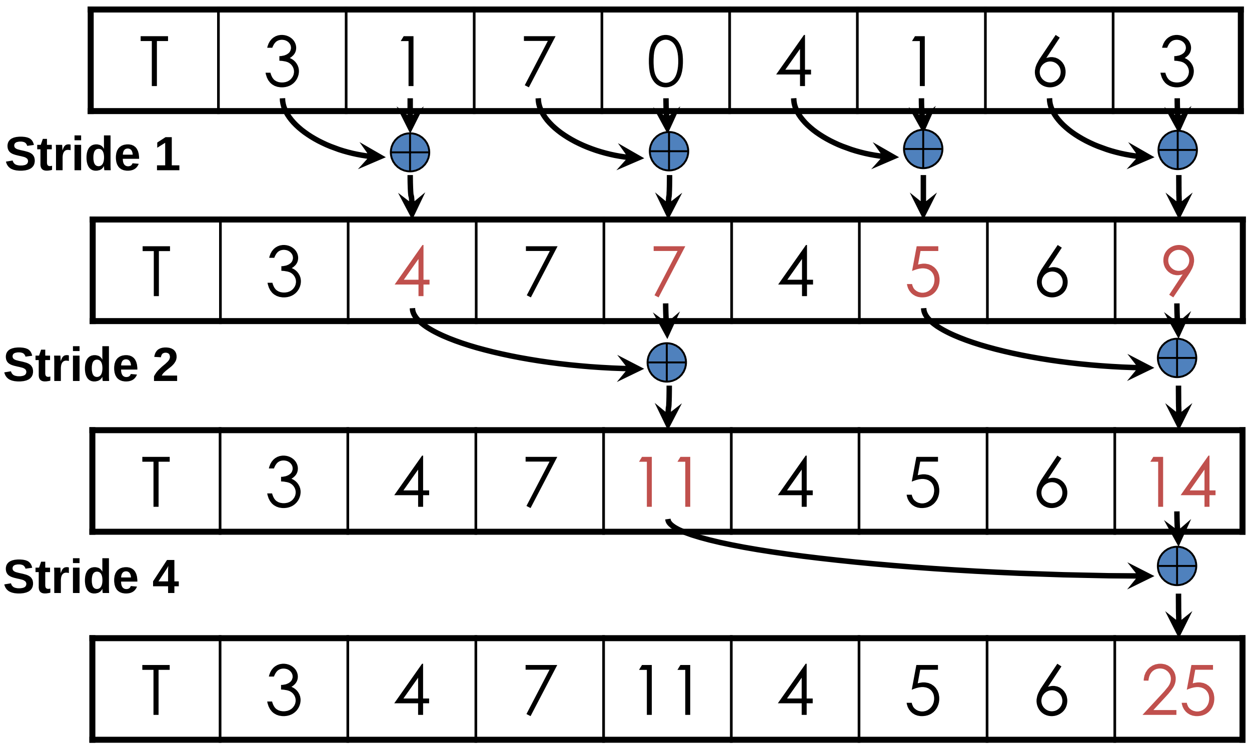


Iteration 2, *n*/4 UEs

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

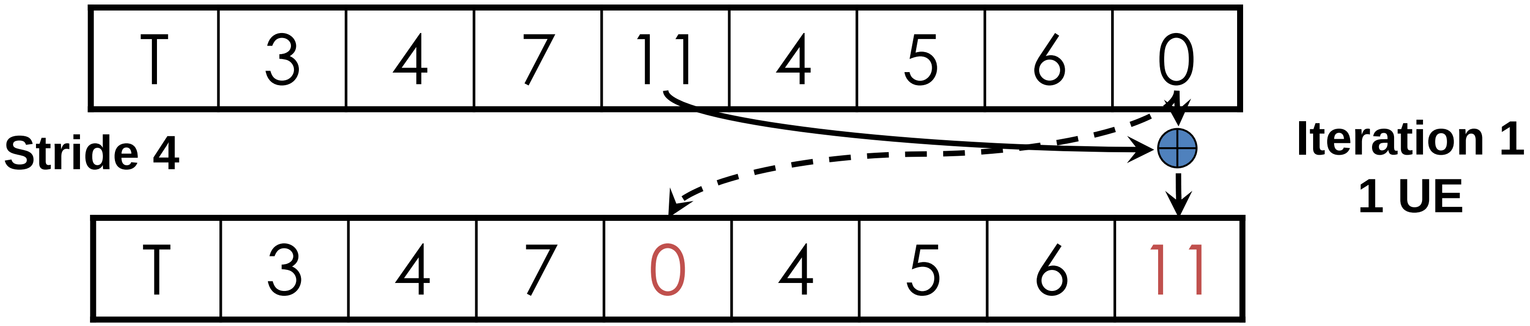
T	3	4	7	11	4	5	6	0

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

T	3	4	7	11	4	5	6	0

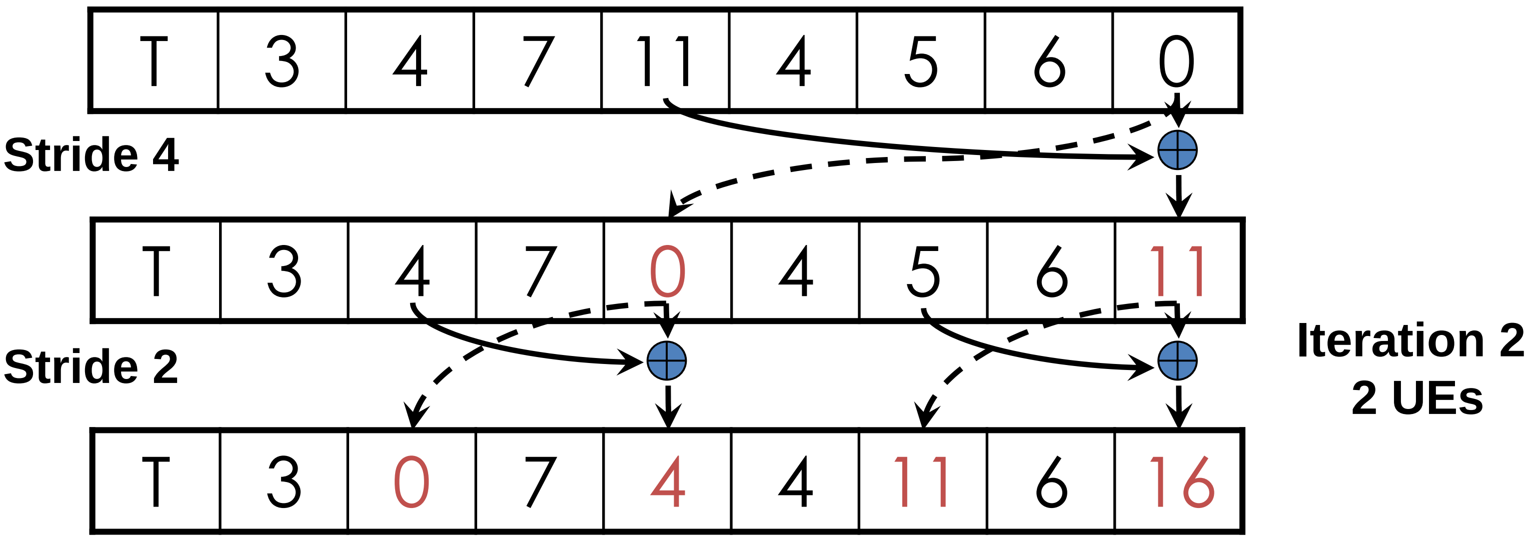
Build Scan From Partial Sums



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.

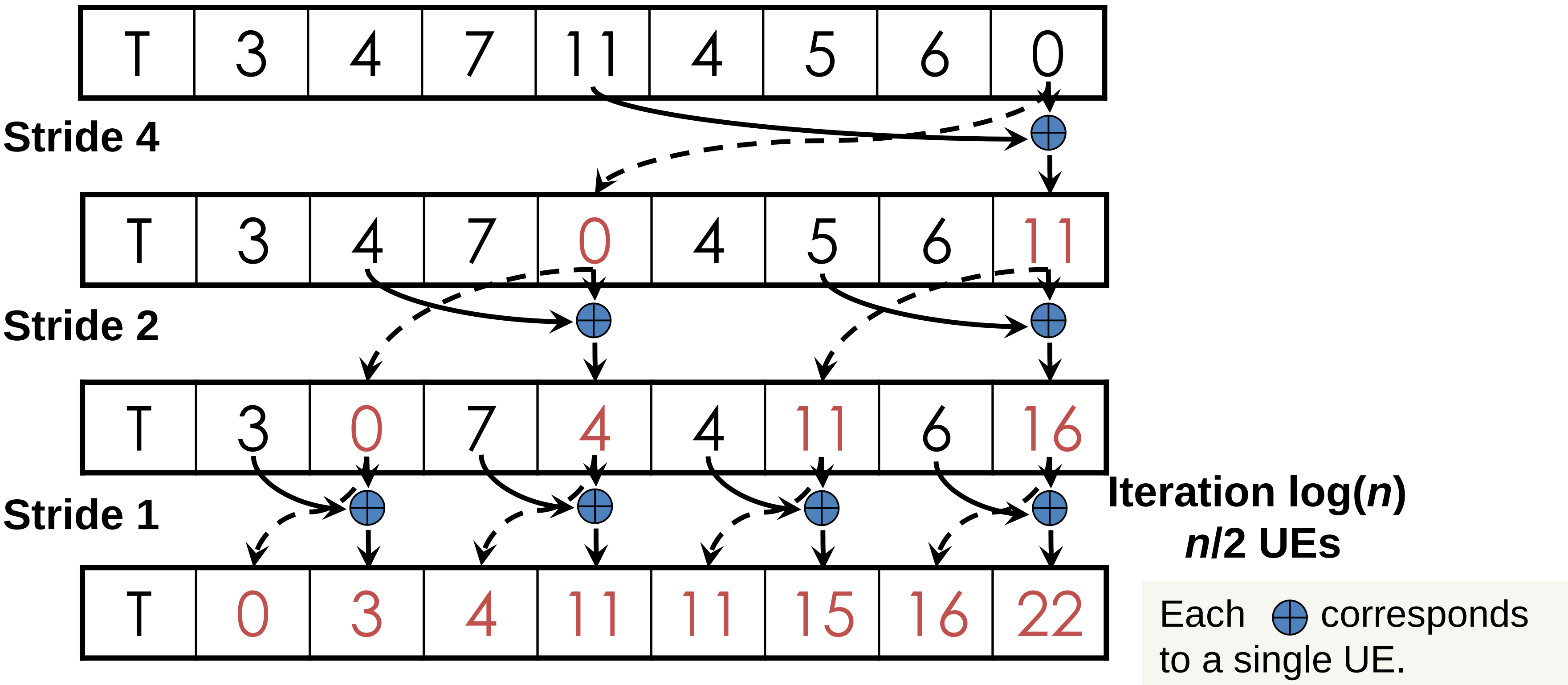
Build Scan From Partial Sums



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!**

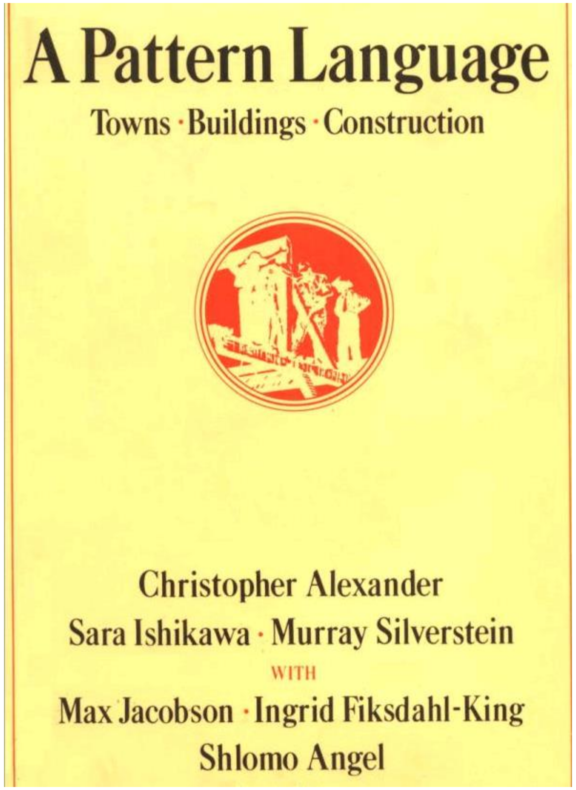
Building Data Structures with Scans

- Fun on the board

History

- Berkeley architecture professor Christopher Alexander

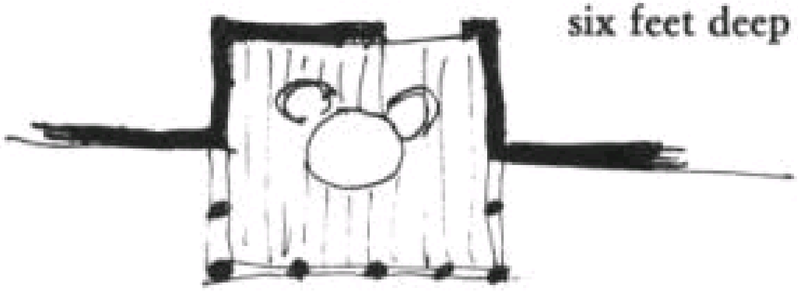
- In 1977, patterns for city planning, landscaping, and architecture in an attempt to capture principles for "living" design



Example 167 (p. 783): 6ft Balcony

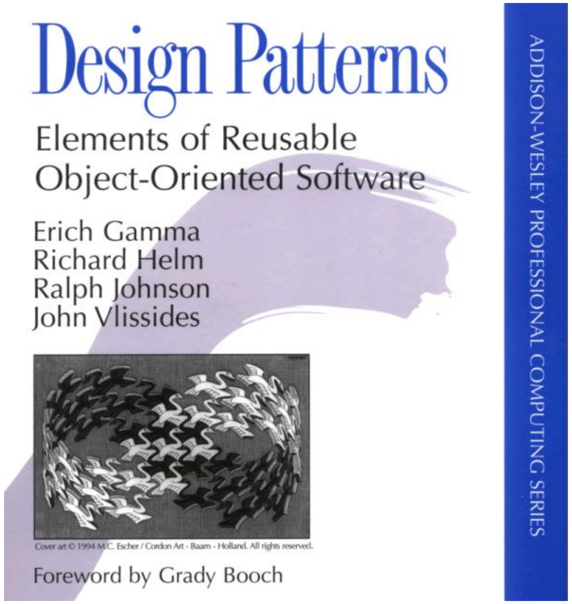
Therefore:

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

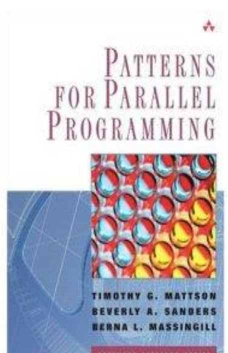
4 Design Spaces

Algorithm Expression

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

Software Construction

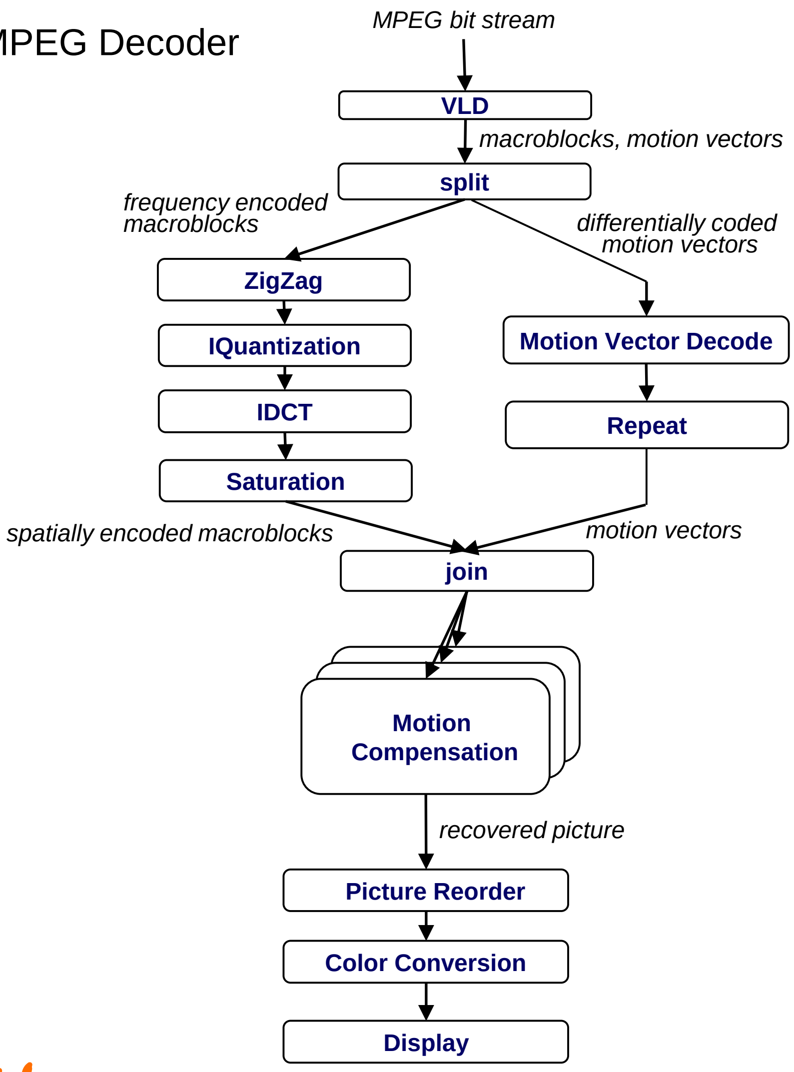
- Supporting Structures
  - Code and data structuring patterns
- Implementation Mechanisms
  - Low level mechanisms used to write parallel programs



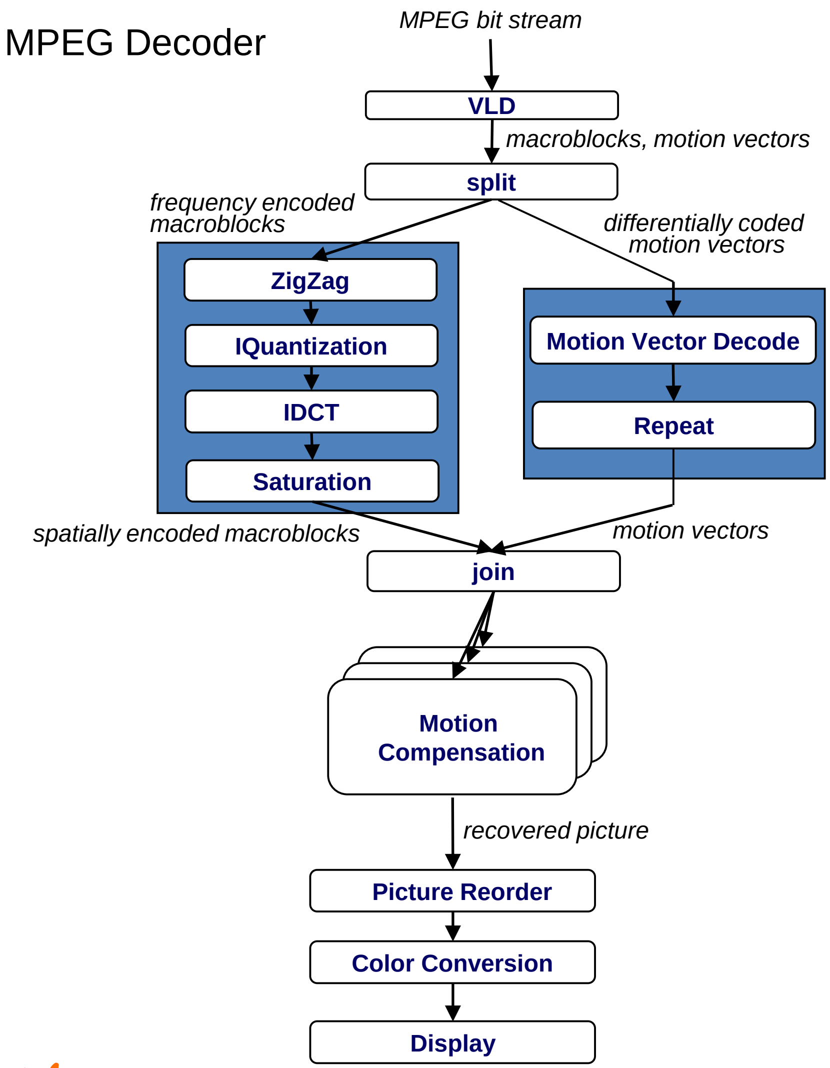
Patterns for Parallel Programming. Mattson, Sanders, and Massingill (2005).

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

MPEG Decoder



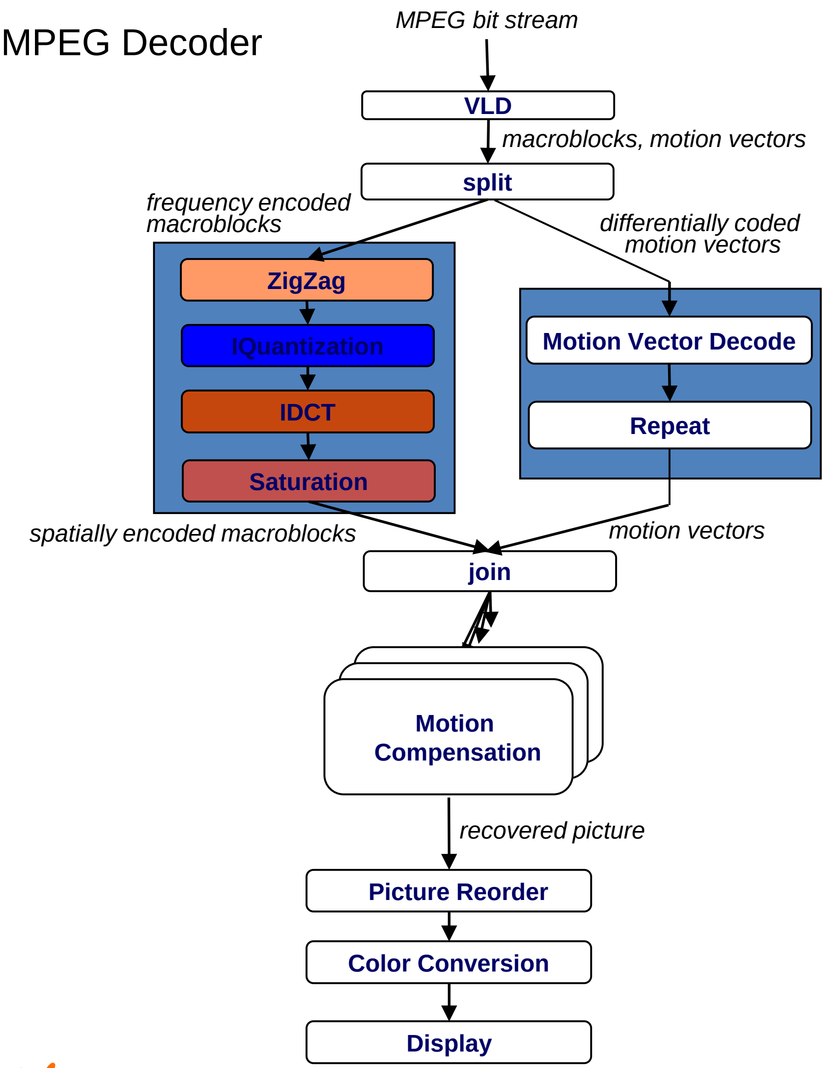
Here's my algorithm. Where's the concurrency?



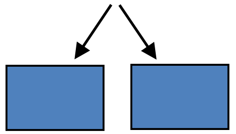
- Task decomposition
  - Independent coarse-grained computation
  - Inherent to algorithm
- Sequence of statements (instructions) that operate together as a group
  - Corresponds to some logical part of program
  - Usually follows from the way programmer thinks about a problem

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

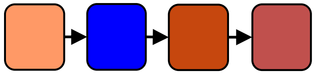
MPEG Decoder



- Task decomposition
  - Parallelism in the application

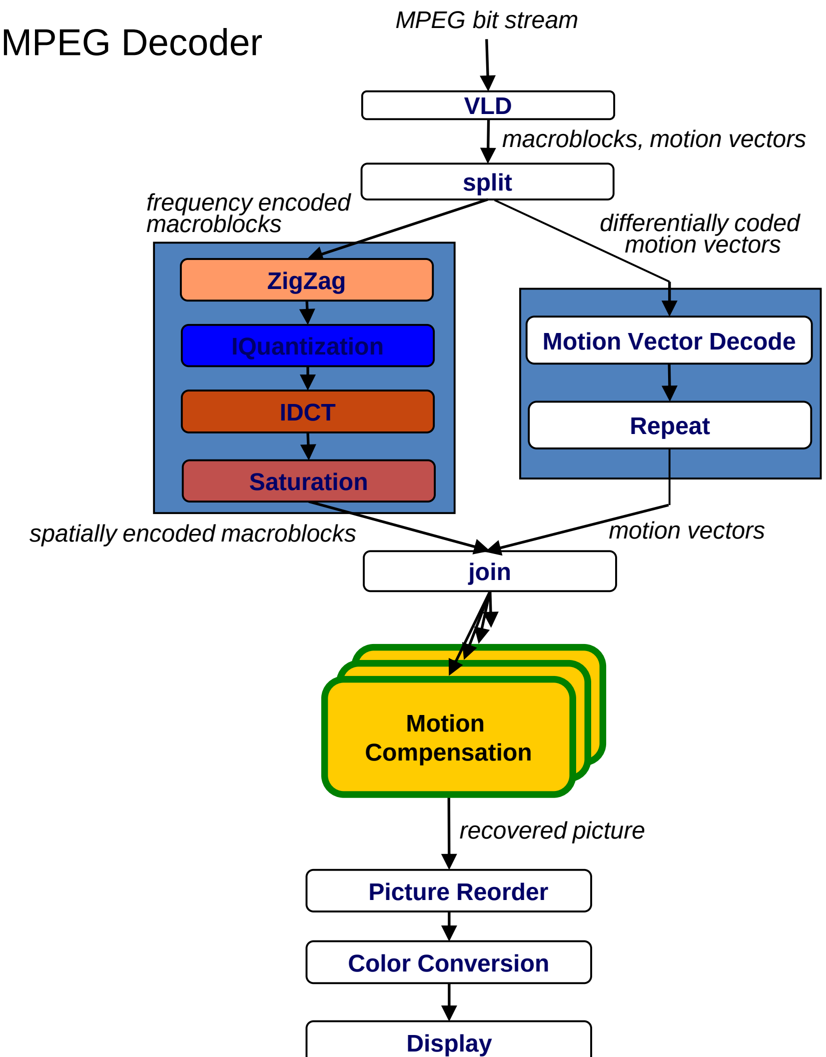


- Pipeline task decomposition
  - Data assembly lines
  - Producer-consumer chains

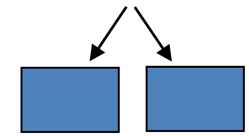


Here’s my algorithm. Where’s the concurrency?

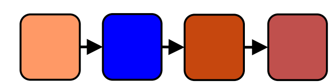
MPEG Decoder



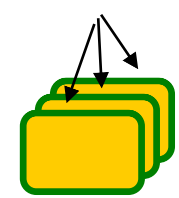
- Task decomposition
  - Parallelism in the application



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



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



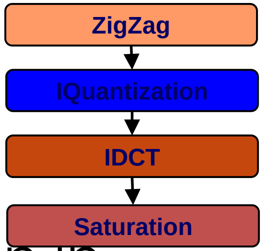
Guidelines for Task Decomposition

- 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



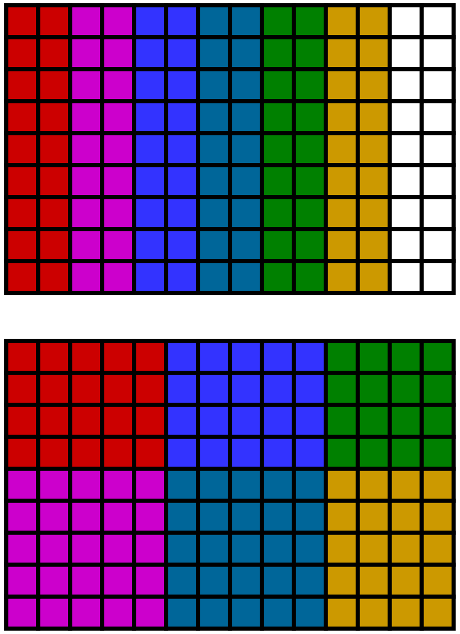
- Data is flowing through a sequence of stages
  - Assembly line is a good analogy
- What's a prime example of pipeline decomposition in computer architecture?
  - Instruction pipeline in modern CPUs
- What's an example pipeline you may use in your UNIX shell?
  - Pipes in UNIX: cat foobar.c | grep bar | wc
- Other examples
  - Signal processing
  - Graphics

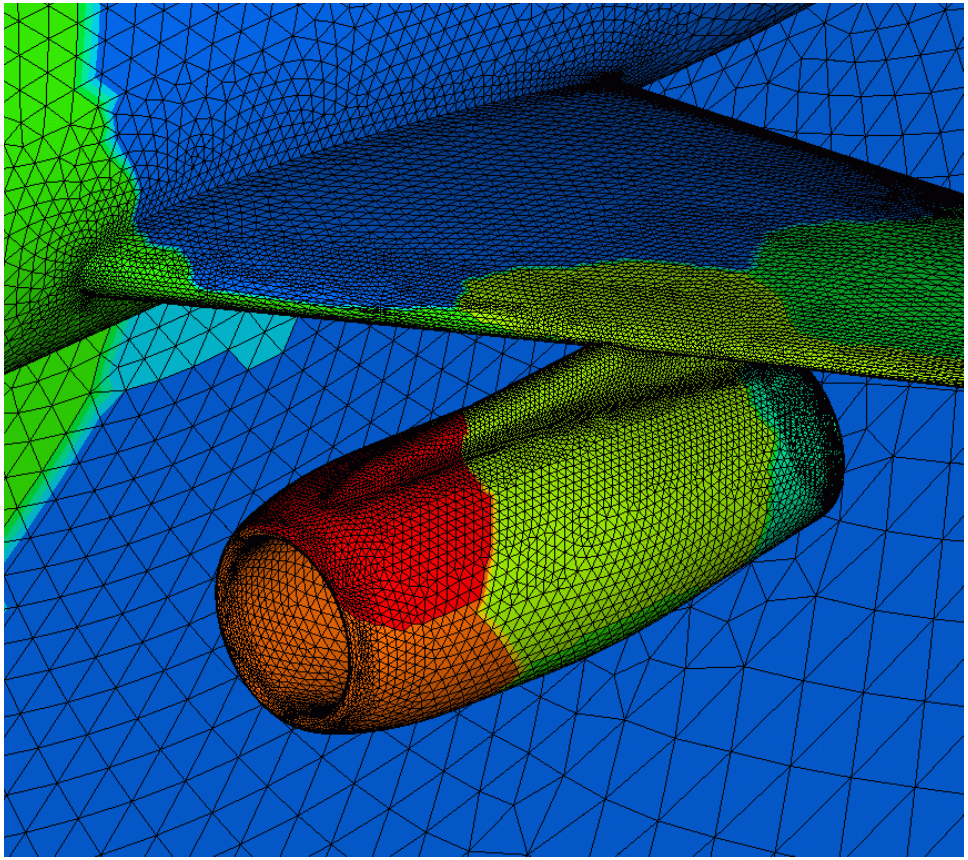
Guidelines for Data Decomposition

- 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

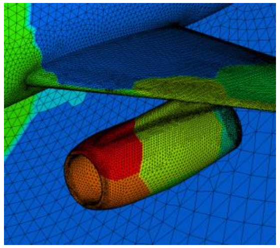
- Geometric data structures
  - Decomposition of arrays along rows, columns, blocks
  - Decomposition of meshes into domains



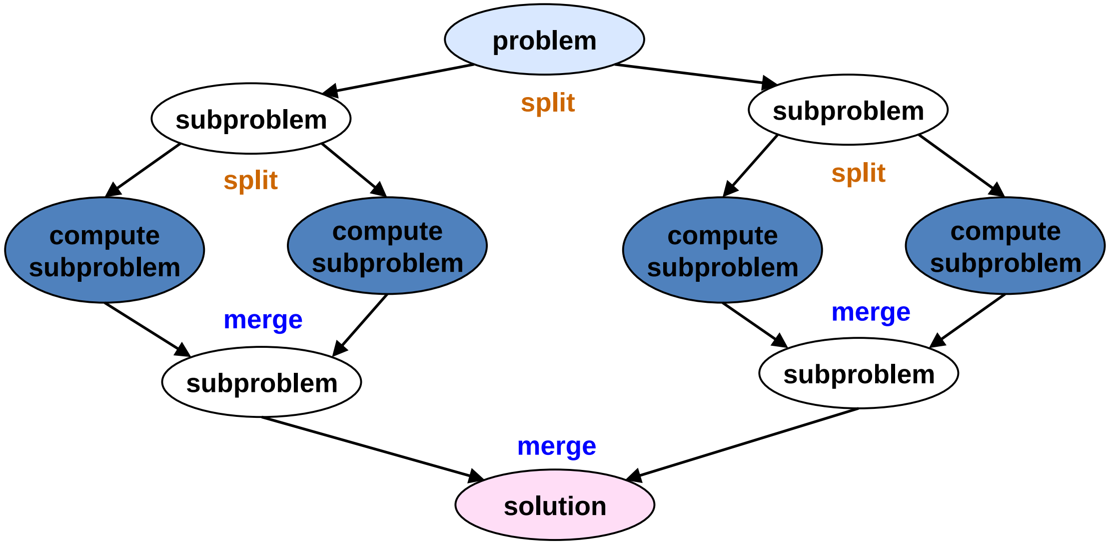


Common Data Decompositions

- 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

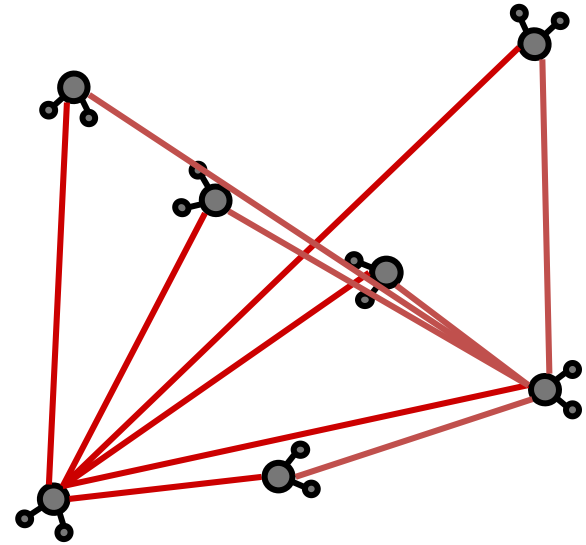


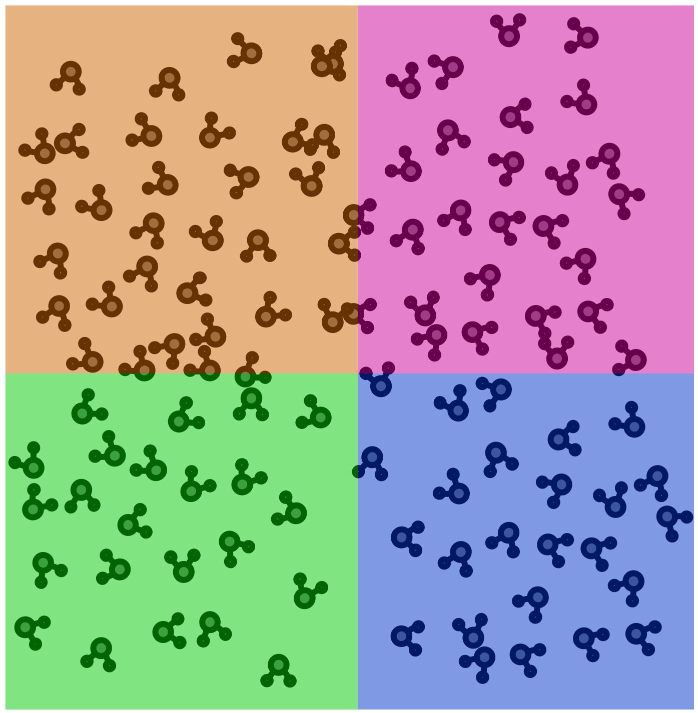
Guidelines for Data Decomposition

- 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

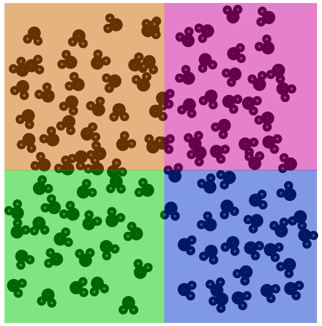
- Molecular dynamics
  - Compute forces
  - Update accelerations and velocities
  - Update positions
- Decomposition
  - Baseline algorithm is N^2
    - All-to-all communication
  - Best decomposition is to treat mols. as a set
  - Some advantages to geometric discussed in future lecture



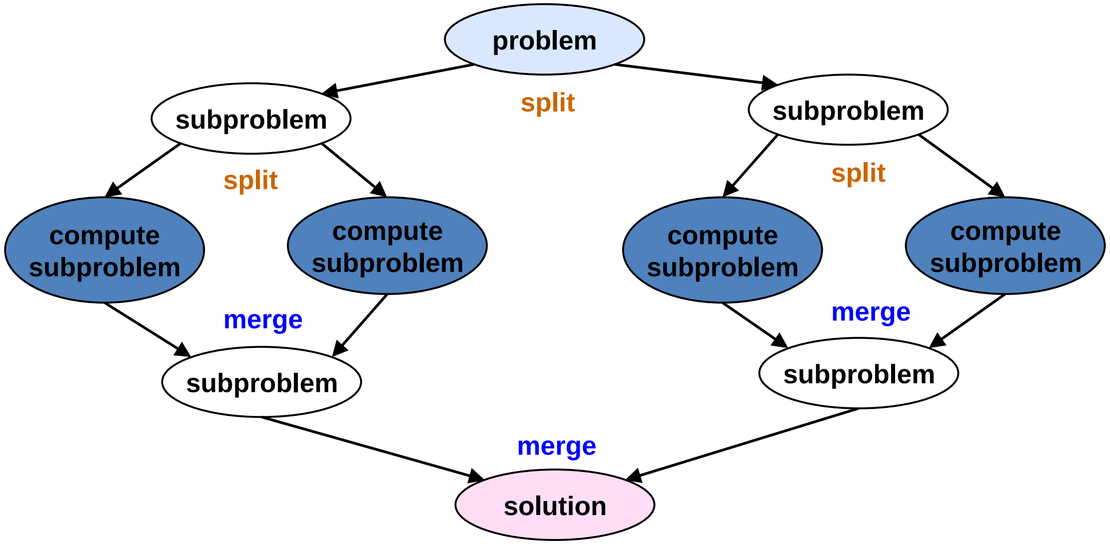


Data Decomposition Examples

- Molecular dynamics
  - Geometric decomposition



- Merge sort
  - Recursive decomposition



Patterns for Parallelizing Programs

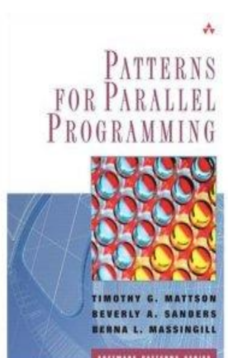
4 Design Spaces

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).

Algorithm Structure Design Space

- Given a collection of concurrent tasks, what's the next step?
- Map tasks to units of execution (e.g., threads)

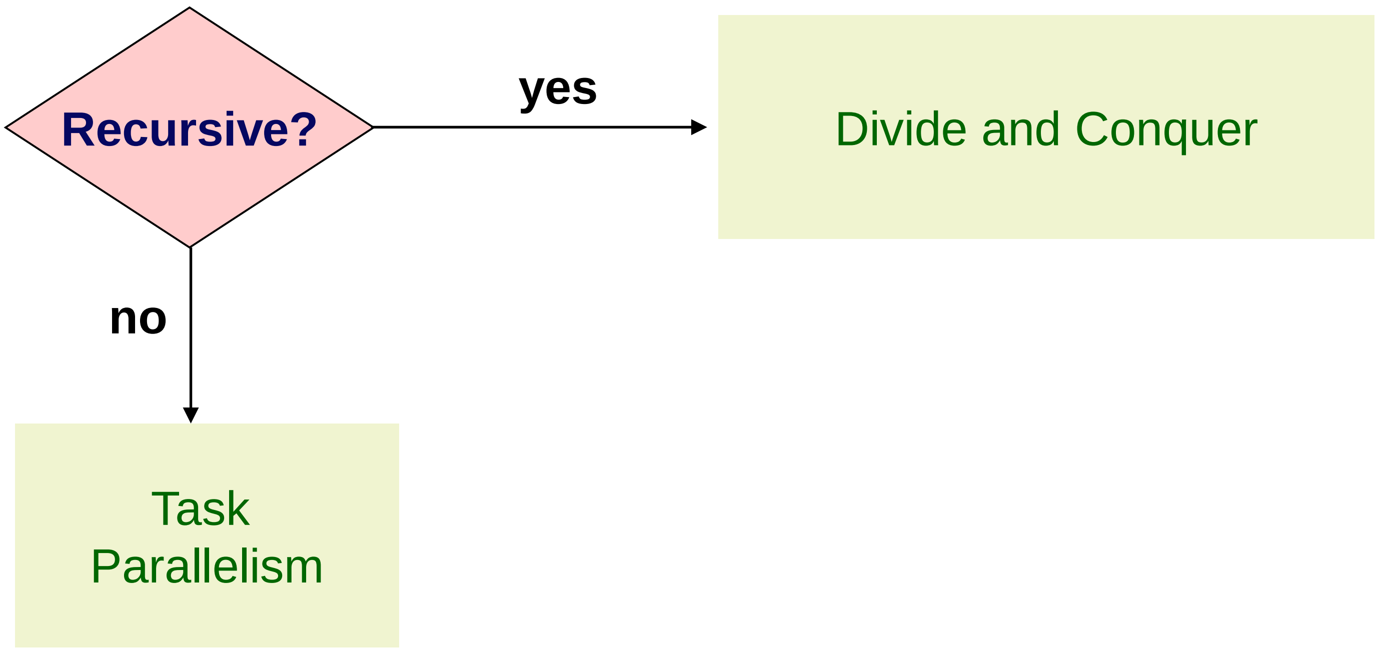
- Important considerations
  - Magnitude of number of execution units platform will support
  - Cost of sharing information among execution units
  - Avoid tendency to over constrain the implementation
    - Work well on the intended platform
    - Flexible enough to easily adapt to different architectures

Major Organizing Principle

- How to determine the algorithm structure that represents the mapping of tasks to units of execution?

- Concurrency usually implies major organizing principle
  - Organize by tasks
  - Organize by data decomposition
  - Organize by flow of data

Organize by Tasks?



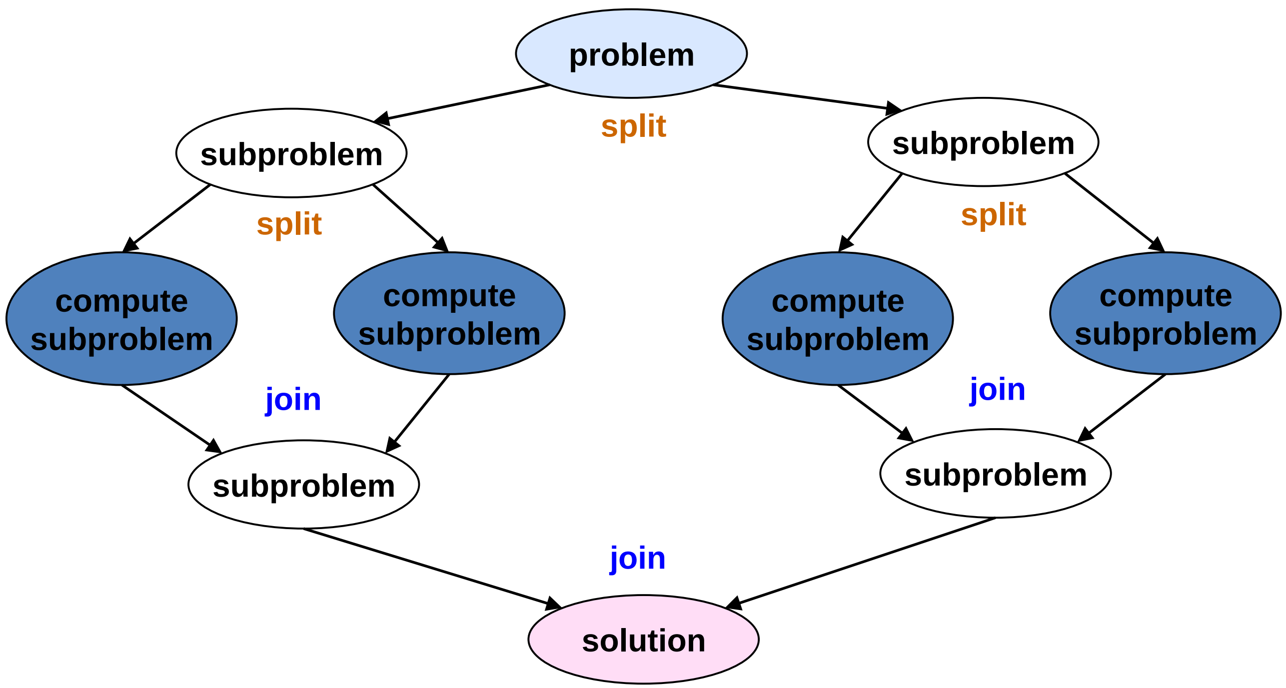
Task Parallelism

- Molecular dynamics
  - Non-bonded force calculations, some dependencies

- Common factors
  - Tasks are associated with iterations of a loop
  - Tasks largely known at the start of the computation
  - All tasks may not need to complete to arrive at a solution

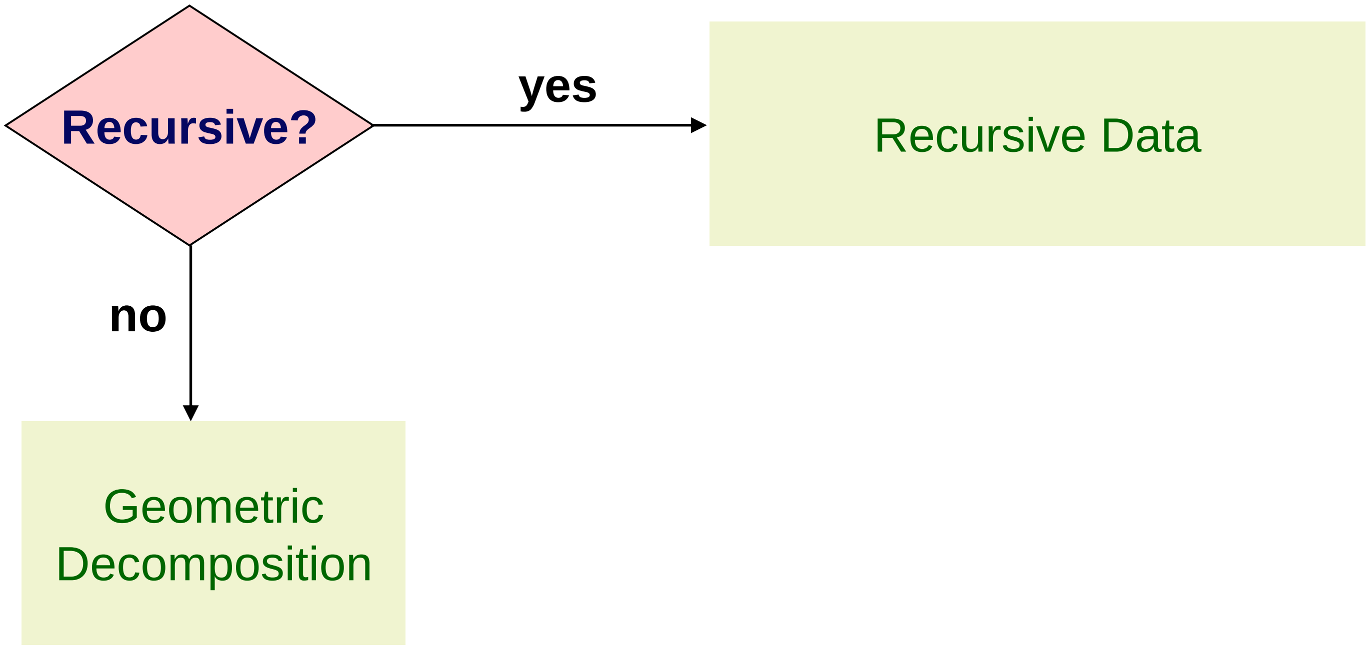
Divide and Conquer

- For recursive programs: divide and conquer
  - Subproblems may not be uniform
  - May require dynamic load balancing



Organize by Data?

- Operations on a central data structure
  - Arrays and linear data structures
  - Recursive data structures



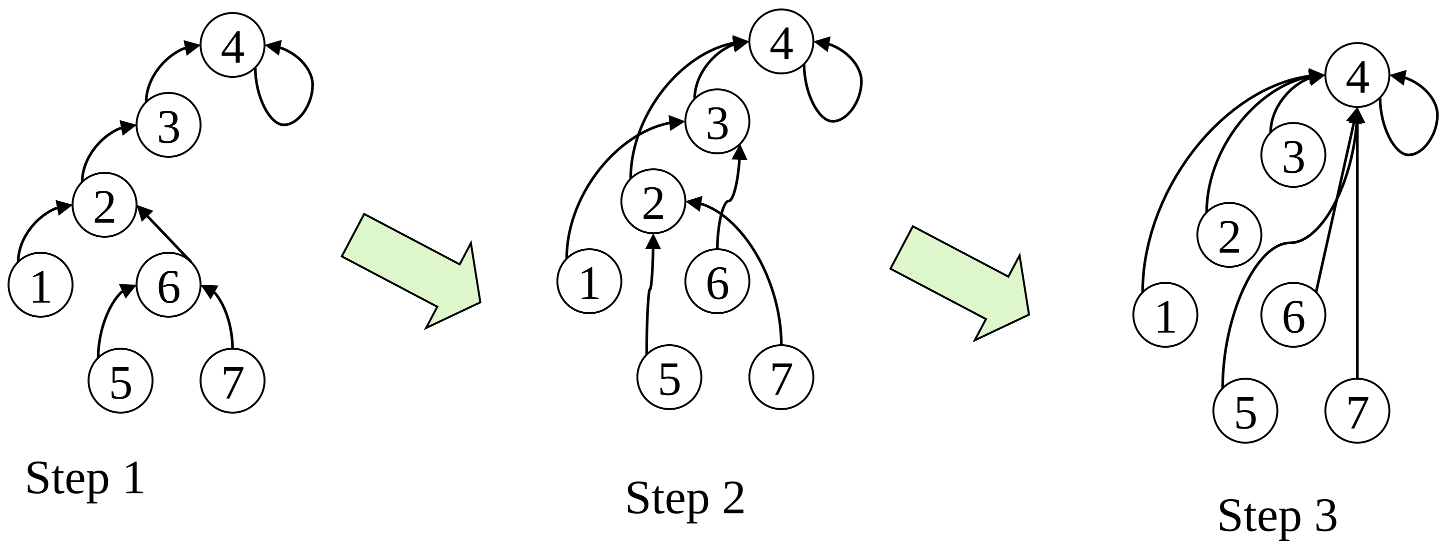
Recursive Data

- Computation on a list, tree, or graph
  - Often appears the only way to solve a problem is to sequentially move through the data structure

- There are however opportunities to reshape the operations in a way that exposes concurrency

Recursive Data Example: Find the Root

- Given a forest of rooted directed trees, for each node, find the root of the tree containing the node
  - Parallel approach: for each node, find its successor's successor, repeat until no changes
    - O(log n) vs. O(n)



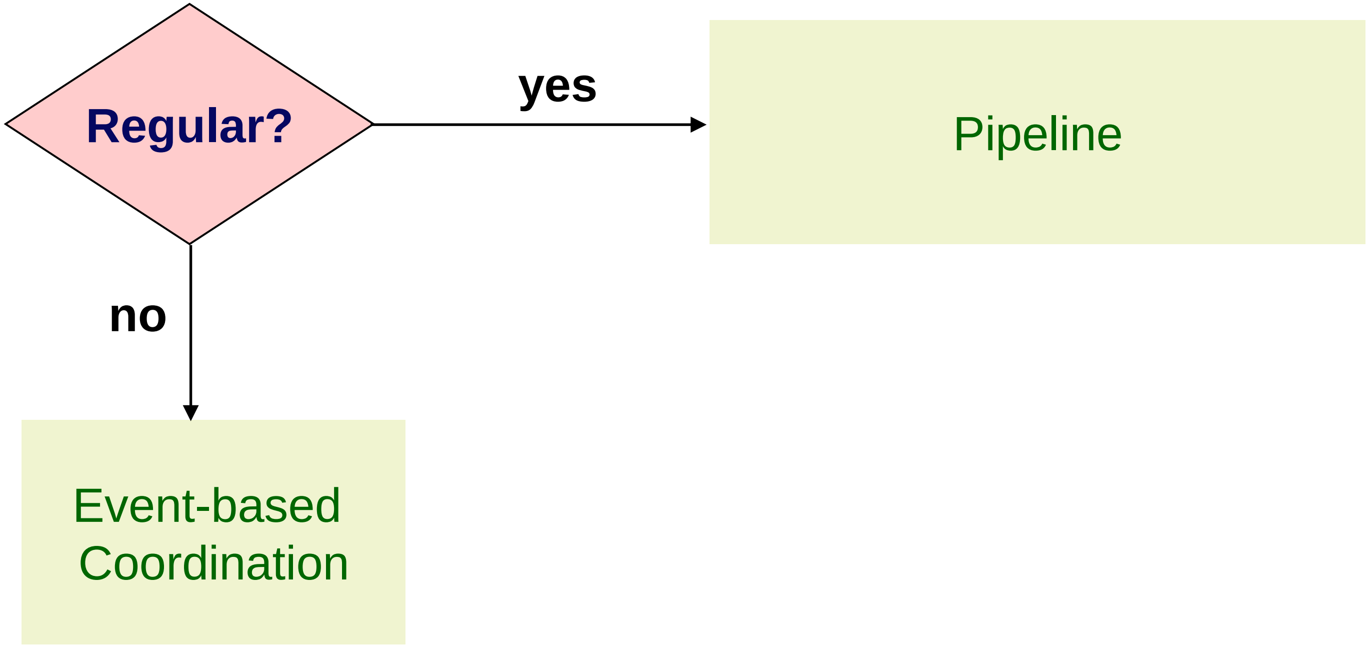
Work vs. Concurrency Tradeoff

- Parallel restructuring of find the root algorithm leads to O(n log n) work vs. O(n) with sequential approach

- Most strategies based on this pattern similarly trade off increase in total work for decrease in execution time due to concurrency

Organize by Flow of Data?

- In some application domains, the flow of data imposes ordering on the tasks
  - Regular, one-way, mostly stable data flow
  - Irregular, dynamic, or unpredictable data flow



Pipeline Throughput vs. Latency

- Amount of concurrency in a pipeline is limited by the number of stages
- Works best if the time to fill and drain the pipeline is small compared to overall running time
- Performance metric is usually the throughput
  - Rate at which data appear at the end of the pipeline per time unit (e.g., frames per second)
- Pipeline latency is important for real-time applications
  - Time interval from data input to pipeline, to data output

Event-Based Coordination

- In this pattern, interaction of tasks to process data can vary over unpredictable intervals
- Deadlocks are a danger for applications that use this pattern
  - Dynamic scheduling has overhead and may be inefficient
    - Granularity a major concern
- Another option is various "static" dataflow models
  - E.g., synchronous dataflow

Patterns for Parallelizing Programs

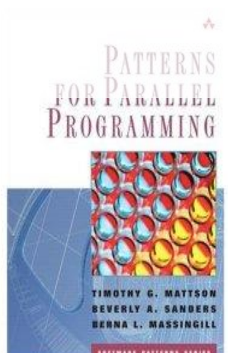
4 Design Spaces

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).

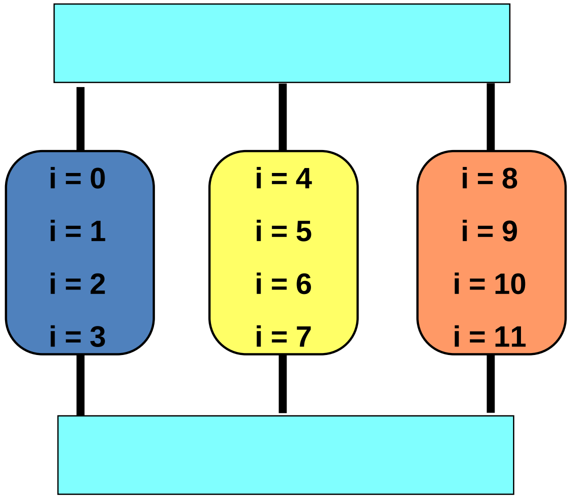
Code Supporting Structures

- Loop parallelism
- Master/Worker
- Fork/Join
- SPMD
- ***Map/Reduce***
- ***Task dataflow***

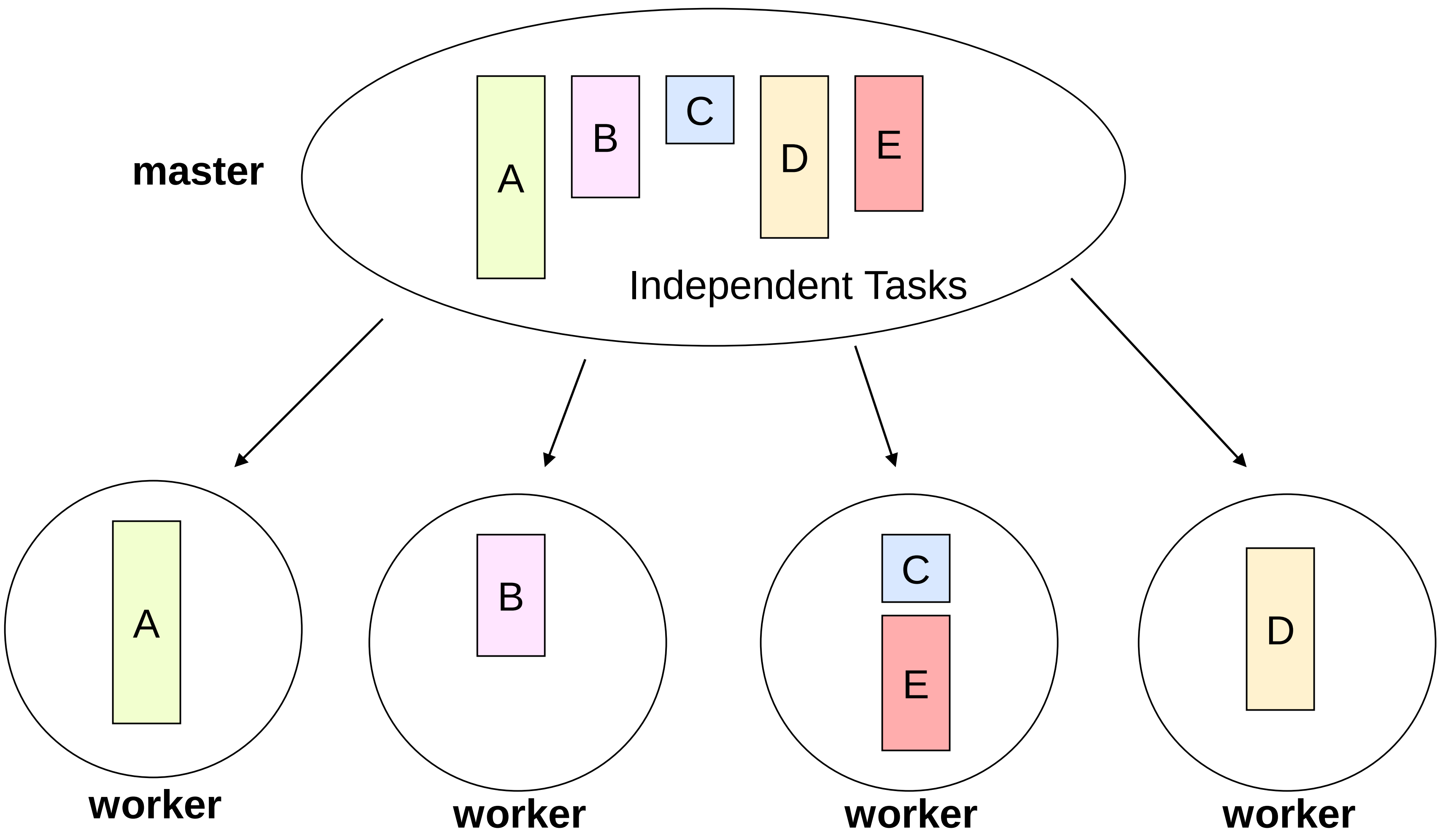
Loop Parallelism Pattern

- Many programs are expressed using iterative constructs
  - Programming models like OpenMP provide directives to automatically assign loop iteration to execution units
  - Especially good when code cannot be massively restructured

```
#pragma omp parallel for
for(i = 0; i < 12; i++)
      C[i] = A[i] + B[i];
```



Master/Worker Pattern



Master/Worker Pattern

- Particularly relevant for problems using task parallelism pattern where task have no dependencies
  - Embarrassingly parallel problems
- Main challenge in determining when the entire problem is complete

Fork/Join Pattern

- Tasks are created dynamically
  - Tasks can create more tasks

- Manages tasks according to their relationship

- Parent task creates new tasks (fork) then waits until they complete (join) before continuing on with the computation

SPMD Pattern

- Single Program Multiple Data: create a single source-code image that runs on each processor
  - Initialize
  - Obtain a unique identifier
  - Run the same program each processor
    - Identifier and input data differentiate behavior
  - Distribute data
  - Finalize

SPMD Challenges

- Split data correctly
- Correctly combine the results
- Achieve an even distribution of the work
- For programs that need dynamic load balancing, an alternative pattern is more suitable

Map/Reduce Pattern

- Two phases in the program
- Map phase applies a single function to all data
  - Each result is a tuple of value and tag
- Reduce phase combines the results
  - The values of elements with the same tag are combined to a single value per tag -- ***reduction***
  - Semantics of combining function are associative
  - Can be done in parallel
  - Can be pipelined with map
- Google uses this for *all* their parallel programs

Communication and Synchronization Patterns

- Communication
  - Point-to-point
  - Broadcast
  - Reduction
  - Multicast
- Synchronization
  - Locks (mutual exclusion)
  - Monitors (events)
  - Barriers (wait for all)
    - Split-phase barriers (separate signal and wait)
      - Sometimes called "fuzzy barriers"
    - Named barriers allow waiting on subset

Quick recap

- Decomposition
  - High-level and fairly abstract
  - Consider machine scale for the most part
  - Task, Data, Pipeline
  - Find dependencies
- Algorithm structure
  - Still abstract, but a bit less so
  - Consider communication, sync, and bookkeeping
  - Task (collection/recursive)
  - Data (geometric/recursive)
  - Dataflow (pipeline/event-based-coordination)
- Supporting structures
  - Loop
  - Master/worker
  - Fork/join
  - SPMD
  - MapReduce

Algorithm Structure and Organization (from the Book)

	Task parallelism	Divide and conquer	Geometric decomposition	Recursive data	Pipeline	Event-based coordination
SPMD	****	***	****	**	***	**
Loop Parallelism	****	**	***			
Master/ Worker	****	**	*	*	****	*
Fork/ Join	**	****	**		****	****

- Patterns can be hierarchically composed so that a program uses more than one pattern

Algorithm Structure and Organization (my view)

	Task parallelism	Divide and conquer	Geometric decomposition	Recursive data	Pipeline	Event-based coordination
SPMD	****	**	****	**	****	*
Loop Parallelism	**** when no dependencies	*	****	*	**** SWP to hide comm.	
Master/ Worker	****	***	***	***	**	****
Fork/ Join	****	****	**	****		*

- Patterns can be hierarchically composed so that a program uses more than one pattern

Patterns for Parallelizing Programs

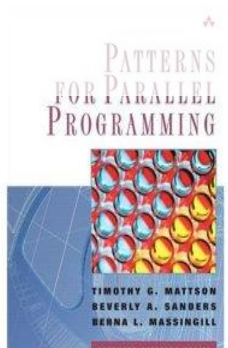
4 Design Spaces

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).

ILP, DLP, and TLP in SW and HW

- ILP
  - OOO
  - Dataflow
  - VLIW
- DLP
  - SIMD
  - Vector
- TLP
  - Essentially multiple cores with multiple sequencers

- ILP
  - Within straight-line code
- DLP
  - Parallel loops
  - Tasks operating on disjoint data
    - No dependencies within parallelism phase
- TLP
  - All of DLP +
  - Producer-consumer chains

ILP, DLP, and TLP and Supporting Patterns

	Task parallelism	Divide and conquer	Geometric decomposition	Recursive data	Pipeline	Event-based coordination
ILP						
DLP						
TLP						

ILP, DLP, and TLP and Supporting Patterns

	Task parallelism	Divide and conquer	Geometric decomposition	Recursive data	Pipeline	Event-based coordination
ILP	inline / unroll	inline	unroll	inline	inline / unroll	inline
DLP	natural or local-conditions	after enough divisions	natural	after enough branches	difficult	local-conditions
TLP	natural	natural	natural	natural	natural	natural

ILP, DLP, and TLP and Implementation Patterns

	SPMD	Loop Parallelism	Mater/Worker	Fork/Join
ILP				
DLP				
TLP				

ILP, DLP, and TLP and Implementation Patterns

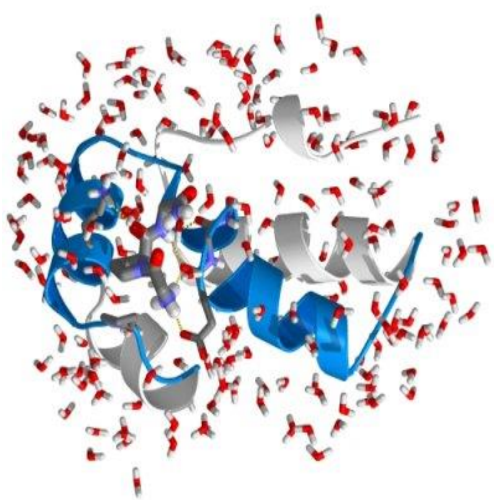
	SPMD	Loop Parallelism	Master/ Worker	Fork/Join
ILP	pipeline	unroll	inline	inline
DLP	natural or local-conditional	natural	local-conditional	after enough divisions + local-conditional
TLP	natural	natural	natural	natural

Outline

- Molecular dynamics example
  - Problem description
  - Steps to solution
    - Build data structures; Compute forces; Integrate for new; positions; Check global solution; Repeat
  - Finding concurrency
    - Scans; data decomposition; reductions
  - Algorithm structure
  - Supporting structures

GROMACS

- Highly optimized molecular-dynamics package
  - Popular code
  - Specifically tuned for protein folding
  - Hand optimized loops for SSE3 (and other extensions)

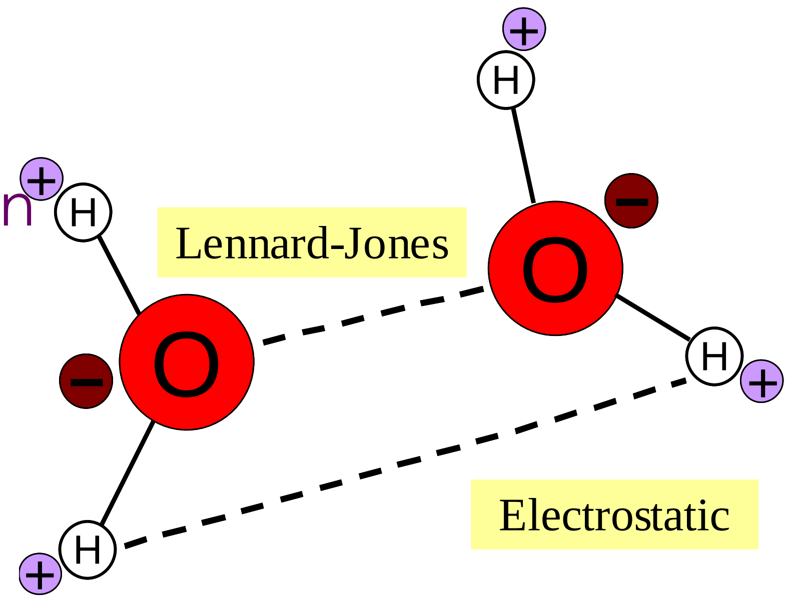


Gromacs Components

- Non-bonded forces
  - Water-water with cutoff
  - Protein-protein tabulated
  - Water-water tabulated
  - Protein-water tabulated
- Bonded forces
  - Angles
  - Dihedrals
- Boundary conditions
- Verlet integrator
- Constraints
  - SHAKE
  - SETTLE
- Other
  - Temperature–pressure coupling
  - Virial calculation

GROMACS Water-Water Force Calculation

- Non-bonded long-range interactions
  - Coulomb
  - Lennard-Jones
  - 234 operations per interaction

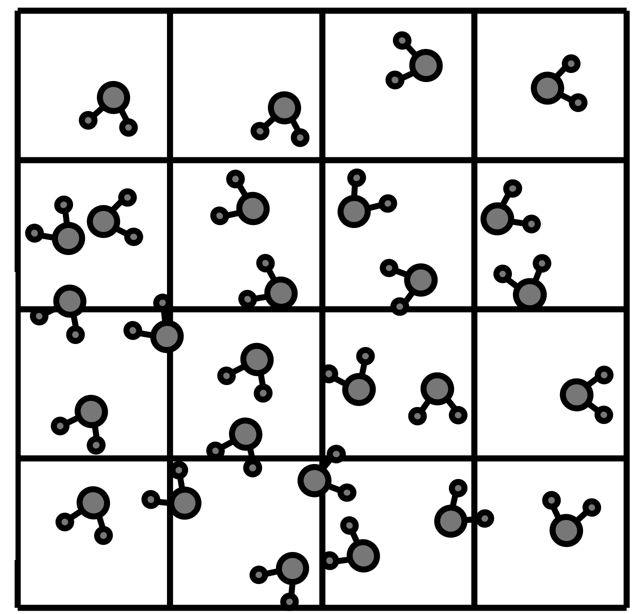


$$V_{nb} = \sum_{i,j} \left[\frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{r_{ij}} + \left(\frac{C_{12}}{r_{ij}^{12}} - \frac{C_6}{r_{ij}^{6}} \right) \right]$$

Water-water interaction ~75% of GROMACS run-time

GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is ***o(n²)***
- GROMACS approximates with a cutoff
  - Molecules located more than r_c apart do not interact
  - ***O(nr_c³)***



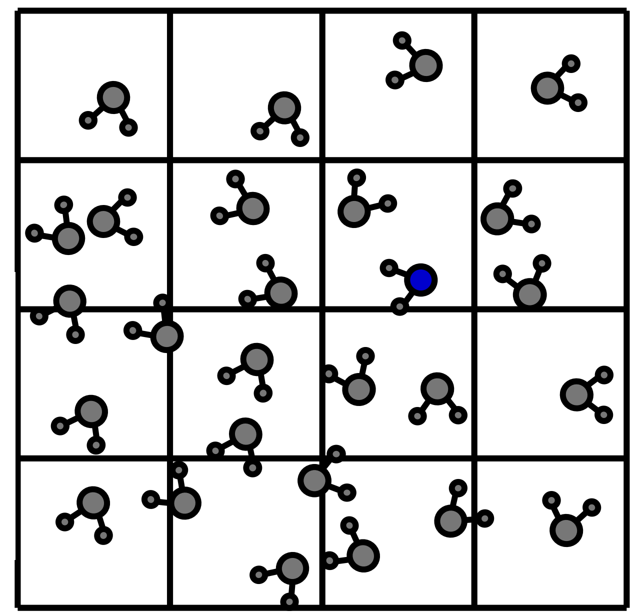
Efficient algorithm leads to variable rate input streams

GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is $\boldsymbol{o(n^2)}$
- GROMACS approximates with a cutoff
  - Molecules located more than $\boldsymbol{r_c}$ apart do not interact
  - $\boldsymbol{O(nr_c^3)}$

central molecules

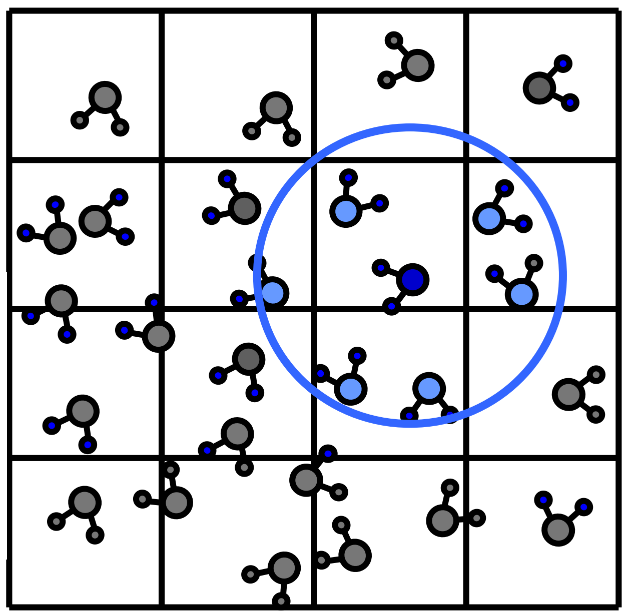
neighbor molecules

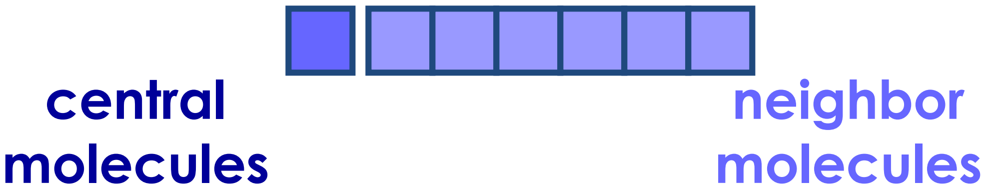


Efficient algorithm leads to variable rate input streams

GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is ***o(n²)***
- GROMACS approximates with a cutoff
  - Molecules located more than r_c apart do not interact
  - $O(nr_c^3)$

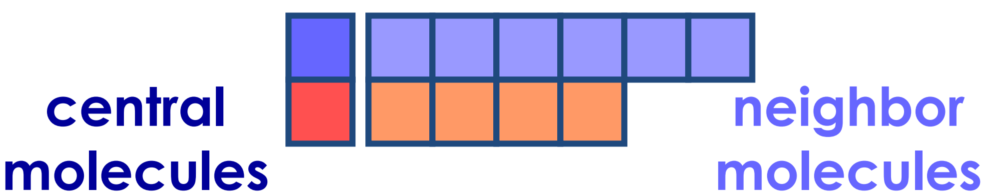


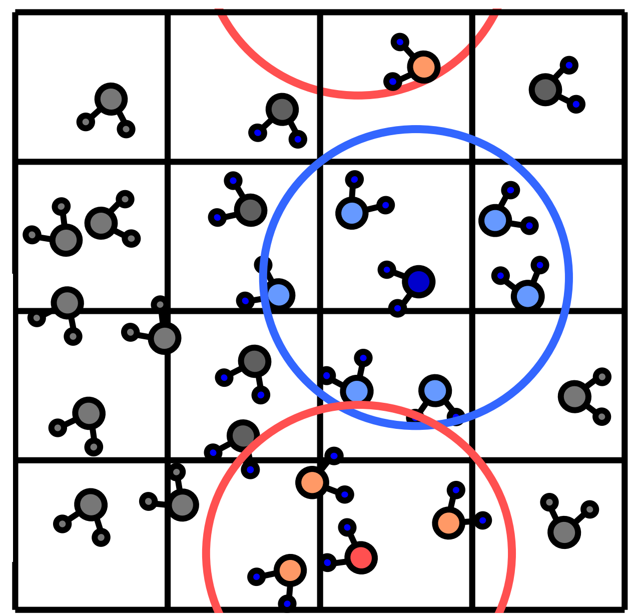


Efficient algorithm leads to variable rate input streams

GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is $o(n^2)$
- GROMACS approximates with a cutoff
  - Molecules located more than r_c apart do not interact
  - $O(nr_c^3)$

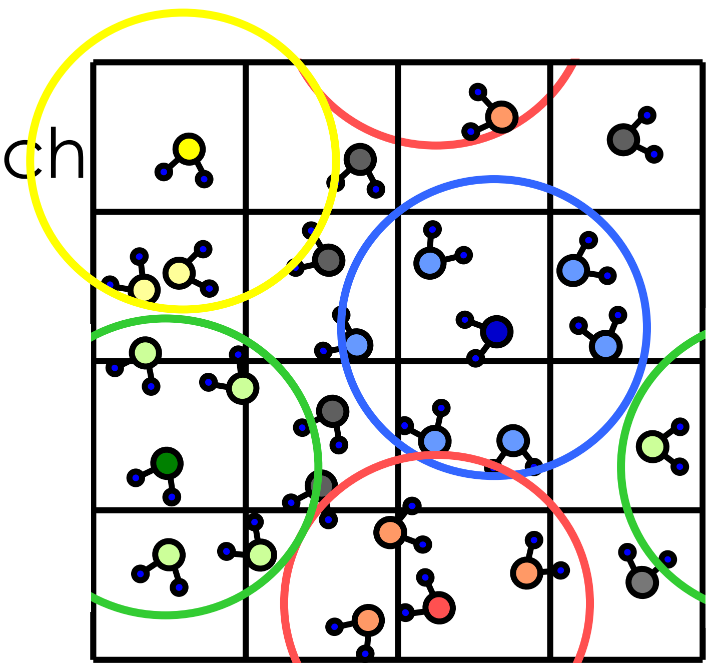


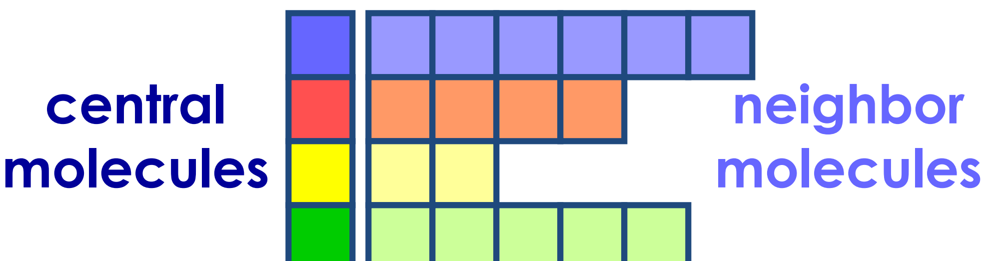


Efficient algorithm leads to variable rate input streams

GROMACS Uses Non-Trivial Neighbor-List Algorithm

- Full non-bonded force calculation is ***o(n²)***
- GROMACS approximates with a cutoff
  - Molecules located more than r_c apart do not interact
  - $O(nr_c^3)$
- Separate neighbor-list for each molecule
  - Neighbor-lists have variable number of elements





Efficient algorithm leads to variable rate input streams

Other Examples

- More patterns
  - Reductions
  - Scans
    - Building a data structure
- More examples
  - Search
  - Sort
  - FFT as divide and conquer
  - Structured meshes and grids
  - Sparse algebra
  - Unstructured meshes and graphs
  - Trees
  - Collections
    - Particles
    - Rays