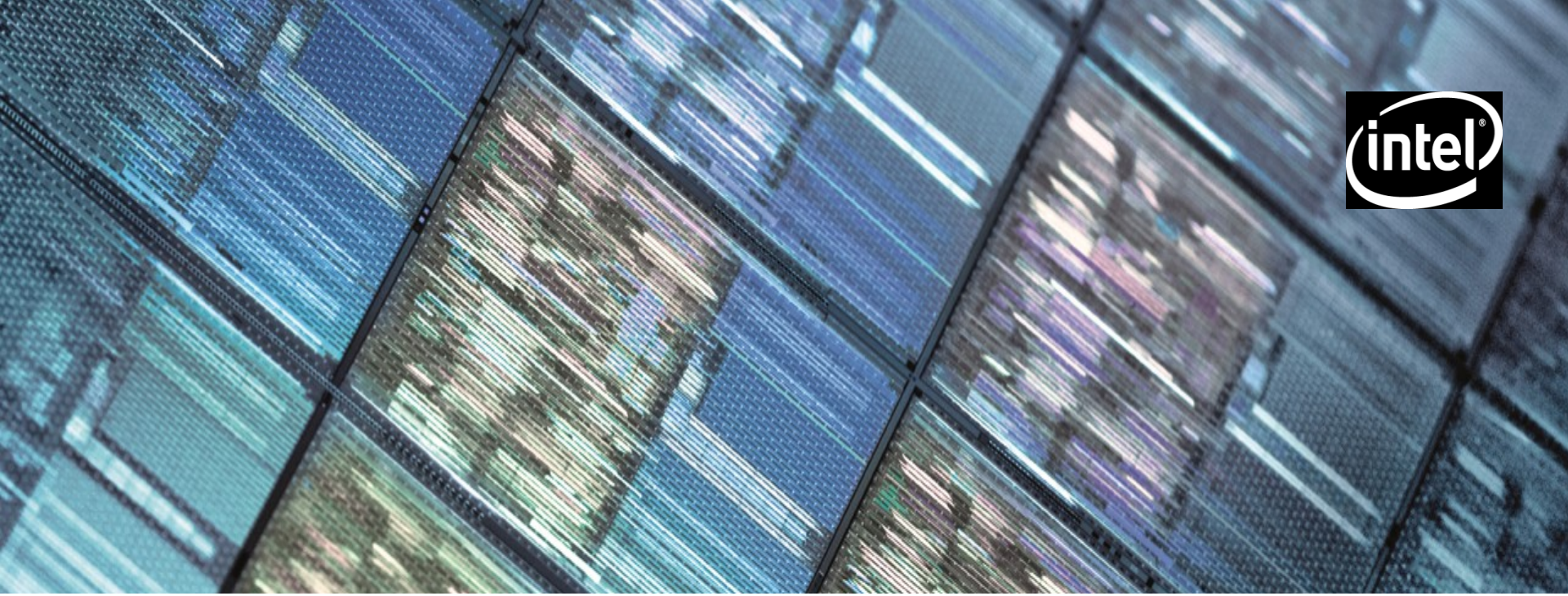


Structured Parallel Programming with Deterministic Patterns

May 14, 2010
USENIX HotPar 2010, Berkeley, California

Michael McCool, Software Architect, Ct Technology
Software and Services Group, Intel Corporation

Patterns

*A **parallel pattern** is a commonly occurring combination of task distribution and data access*

- Many common programming models support either only a small number of patterns, or only low-level hardware mechanisms
  - So often common patterns implemented only as "conventions"

Observation: a small number of patterns, most of them deterministic, can support a wide range of applications

Thesis: A system that ***directly*** supports these deterministic patterns and allows their composition can generate efficient implementations on a variety of hardware architectures

Motivation for Pattern-based Design

Deterministic patterns → higher maintainability

- No need to debug race conditions if it not possible to create them
- Allow introduction of races only where necessary, and limit scope
- Determinism and consistency with single serial execution order simplifies user understanding, debugging and testing

Application oriented patterns → higher productivity

- Patterns derived from common use cases in applications
  - Subset of patterns are universal: gives wide applicability
  - Patterns can also target specific domains:
    - Makes simple things simple
- Patterns encourage high-level reasoning
  - Focus users on what really matters: parallelism and data locality
  - Simplifies learning how to write efficient programs

Serial Patterns

*The following patterns are the basis of "**structured programming**" for serial computation:*

- Sequence
- Selection
- Iteration
- Recursion
- Random read
- Random write
- Stack allocation
- Heap allocation
- Objects/closures

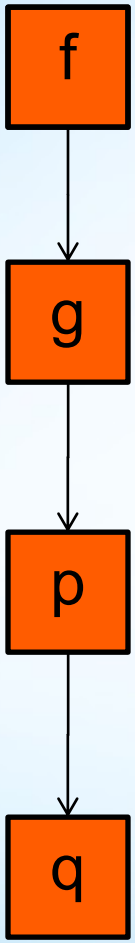
Compositions of control flow patterns can be used in place of unstructured mechanisms such as "goto."

Parallel Patterns

*The following additional parallel patterns can be used for "**structured parallel programming**":*

- Superscalar sequence
- Speculative selection
- Map
- Recurrence/scan
- Reduce
- Pack/expand
- Nest
- Pipeline
- Partition
- Stencil
- Search/match
- Gather
- **Permutation scatter*
- **Merge scatter*
- *!Atomic scatter*
- Priority scatter

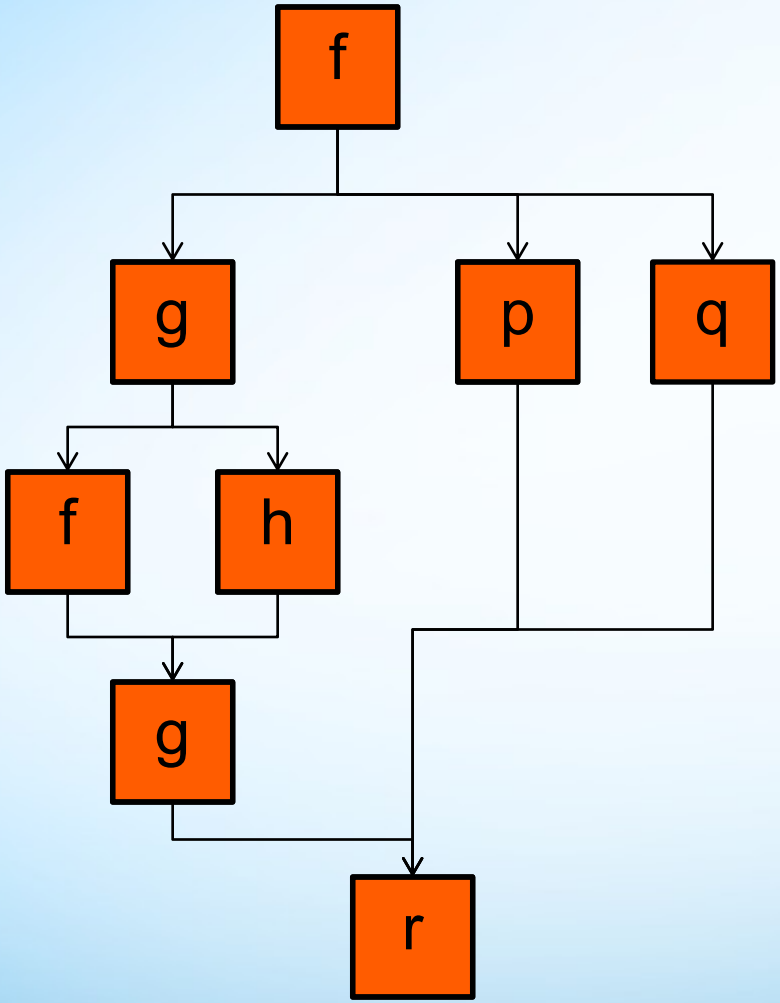
Sequence



A serial sequence is executed in the exact order given:

```
B = f(A);
C = g(B);
E = p(C);
F = q(A);
```

Superscalar Sequence

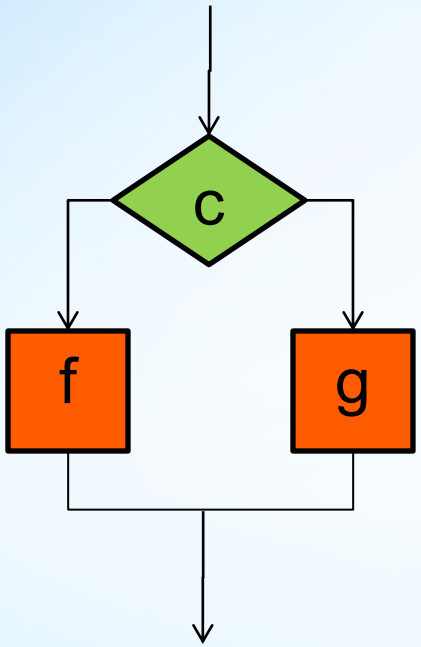


Developer writes "serial" code:

```
B = f(A);
C = g(B);
E = f(C);
F = h(C);
G = g(E,F);
P = p(B);
Q = q(B);
R = r(G,P,Q);
```

- However, tasks only need to be ordered by data dependencies
- Depends on limiting scope of data dependencies

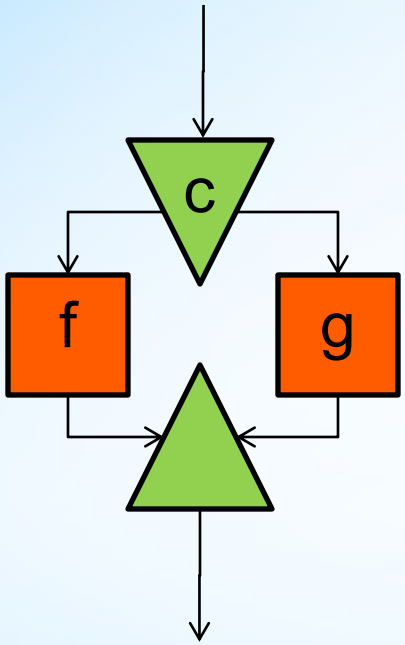
Selection



The condition is evaluated first, then one of two tasks is executed based on the result.

```
IF (c) {
    f
} ELSE {
    g
}
```

Speculative Selection



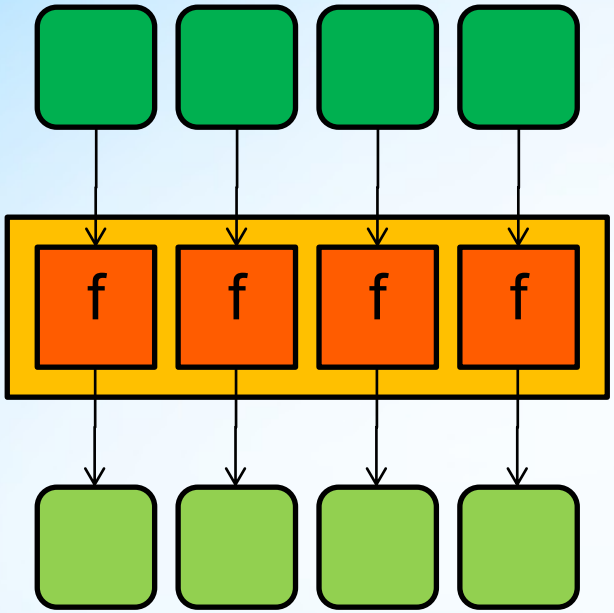
Both sides of a conditional and the condition are evaluated in parallel, then the unused branch is cancelled.

```
SELECT (c) {
    f
} ELSE {
    g
}
```

- Effort in cancelled task "wasted"
- Use only when a computational resource would otherwise be idle, or tasks are on critical path

Examples: collision culling; ray tracing; clipping; discrete event simulation; search

Map



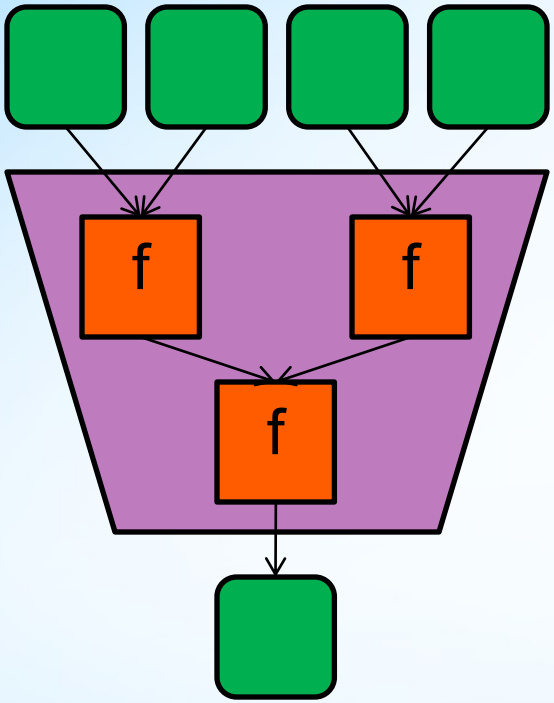
- Map replicates a function over every element of an index set (which may be abstract or associated with the elements of an array).

```
A = map(f,B);
```

- This replaces one specific usage of iteration in serial programs: processing every element of a collection with an independent operation.

Examples: gamma correction and thresholding in images; color space conversions; Monte Carlo sampling; ray tracing.

Reduction



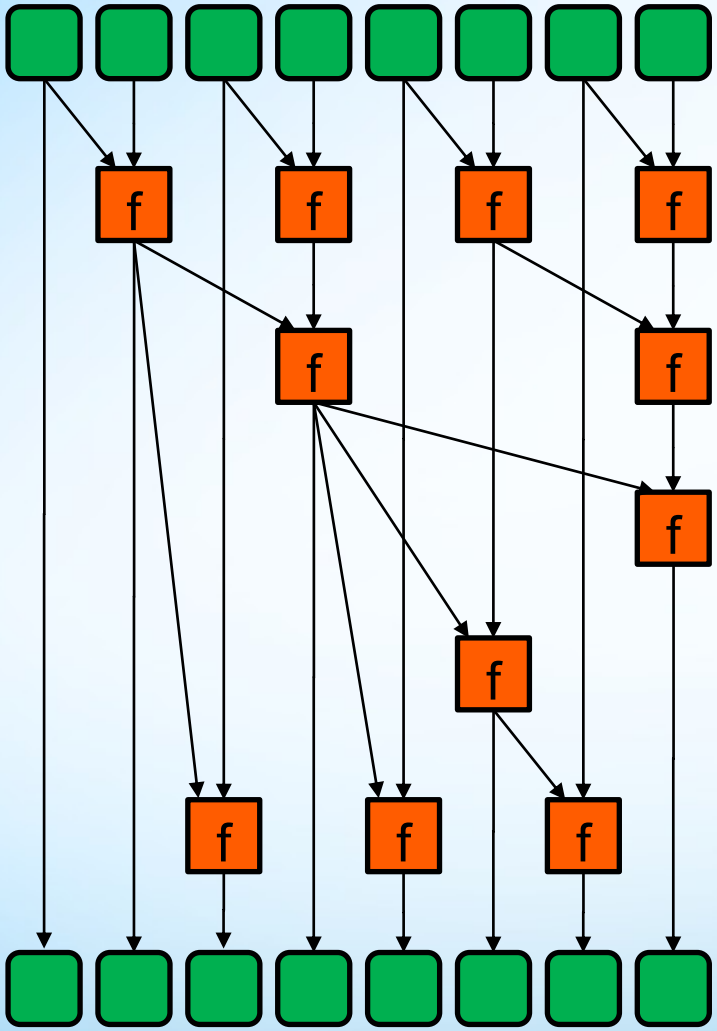
- Reduce combines every element in a collection into one element using an associative operator.

```
b = reduce(f,B);
```

- For example, reduce can be used to find the sum or maximum of an array.
- There are some variants that arise from combination with *partition* and *search*

Examples: averaging of Monte Carlo samples; convergence testing; image comparison metrics; sub-task in matrix operations.

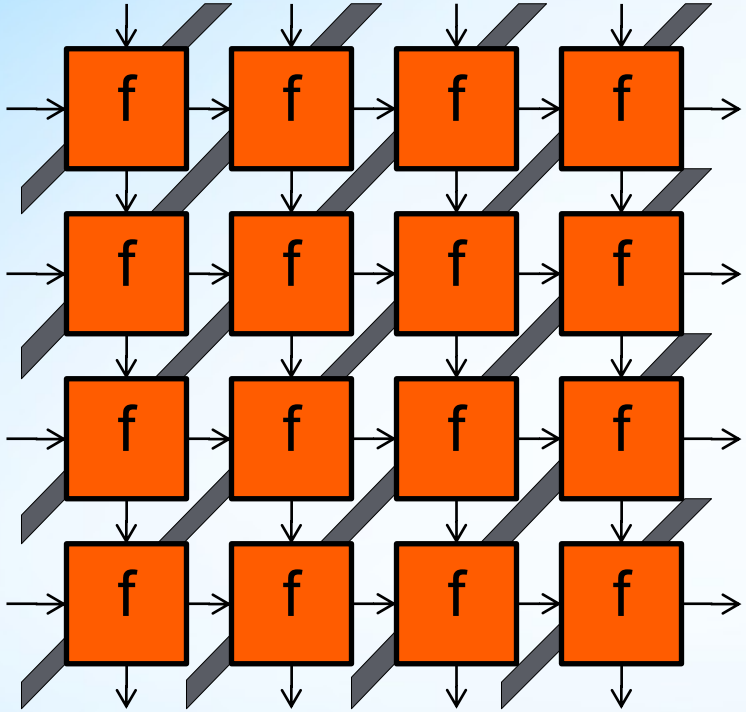
Scan



- Scan computes *all* partial reductions
- Allows parallelization of many 1D recurrences
- Requires an associative operator
- Requires 2n work over serial execution, but lg n steps

Examples: integration, sequential decision simulations in financial engineering, can also be used to implement pack

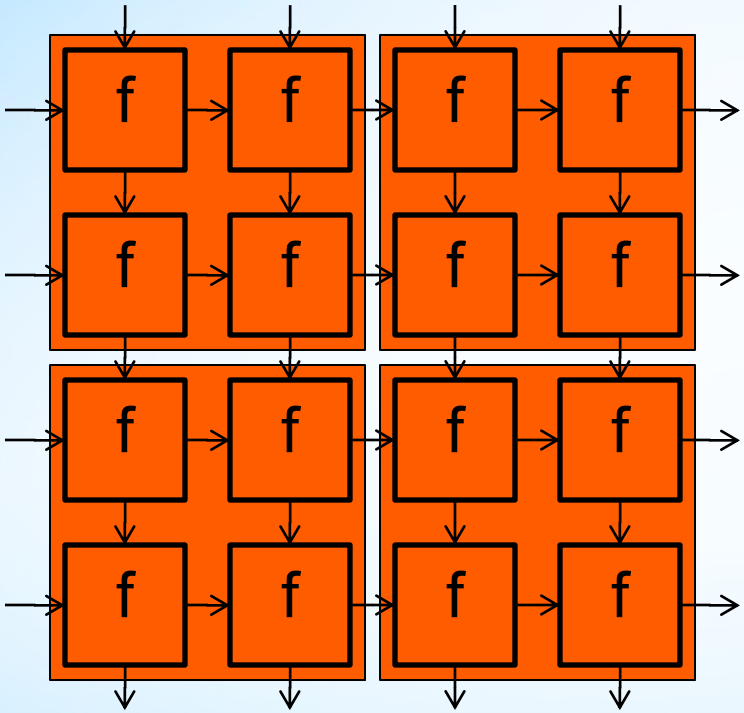
Recurrences



Examples: infinite impulse response filters; sequence alignment (Smith-Waterman dynamic programming); matrix factorization

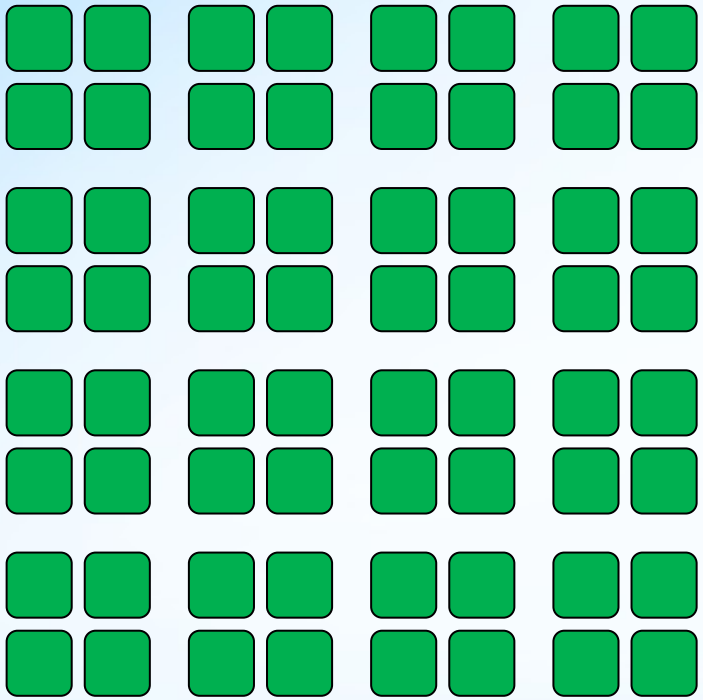
- Recurrences arise from the data dependency pattern given by *nested* loop-carried dependencies.
  - nD recurrences can *always* be parallelized over n-1 dimensions by Lamport's hyperplane theorem
- Execution of parallel slices can be performed either via iterative map or via wavefront parallelism

Recurrences: Implementation Note



- Implementation can use blocking for higher performance
- When combined with the "pipeline" pattern recurrences implements "wavefront" computation
- Can also be combined with superscalar execution (see recent ICS paper…)

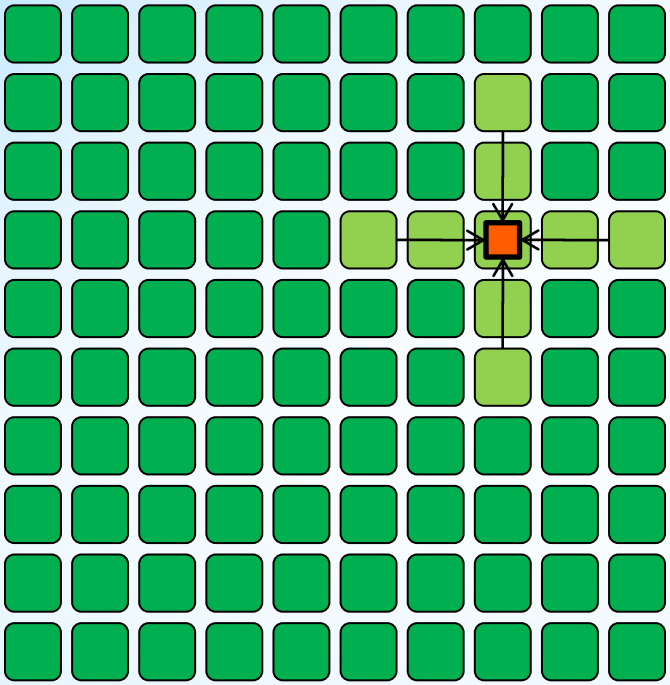
Partition



Examples: JPG and other macroblock compression; divide-and-conquer matrix multiplication; coherency optimization for cone-beam reconstruction

- Partition breaks an input collection into a collection of collections
- Useful for divide-and-conquer algorithms
- Variants:
  - Uniform: dice
  - Non-uniform: segment
  - Overlapping: tile
- Issues:
  - How to deal with boundary conditions?
- Partitions don't move data, they just provide an alternative "view" of its organization

Stencil

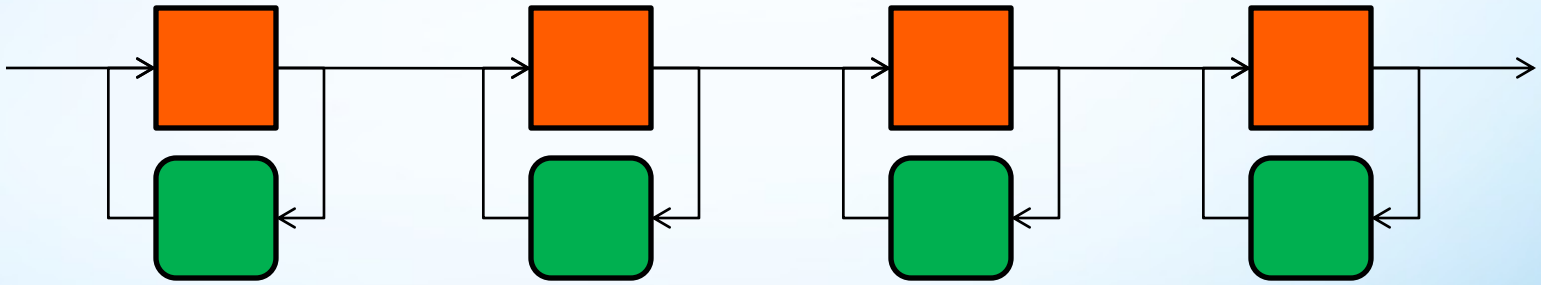


- Apply function to all neighbourhoods of an array
- Neighbourhoods given by set of relative offsets
- Optimized implementation requires blocking and sliding windows
- "Boundary modes" on array accesses useful

Examples: image filtering including convolution, median, anisotropic diffusion; simulation including fluid flow, electromagnetic, and financial PDE solvers, lattice QCD

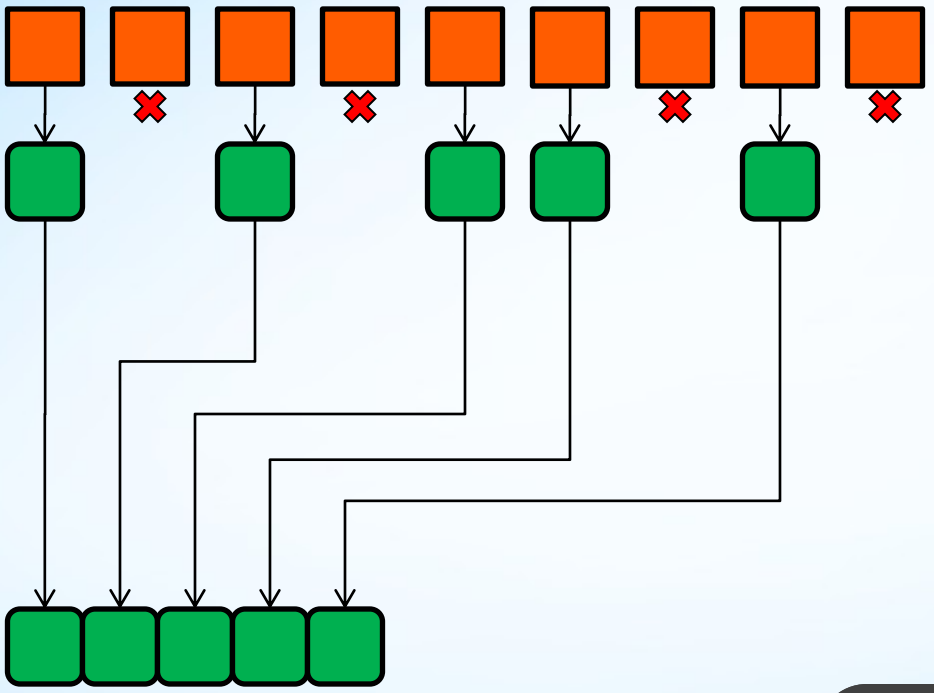
Pipeline

- Tasks can be organized in chain with *local state*
- Useful for serially dependent tasks like codecs
- Whole chain applied like map to collection or *stream*
- Implementation of many sub-patterns may be optimized for pipeline execution when inside this pattern



Examples: codecs with variable-rate compression; video processing; spam filtering.

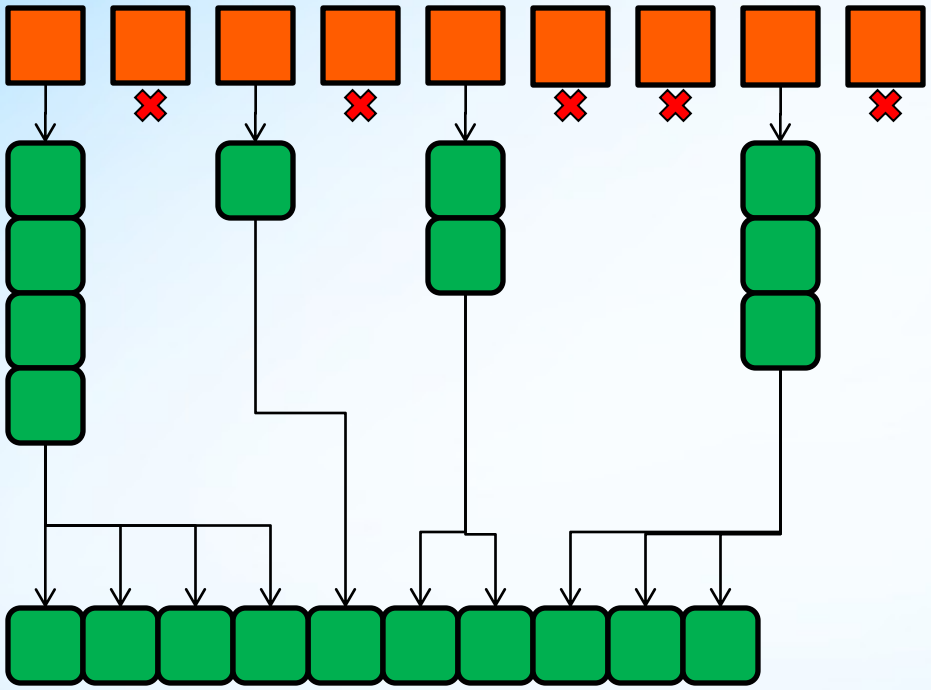
Pack



- Pack allows deletion of elements from a collection and elimination of unused space
- Useful when fused with map and other patterns to avoid unnecessary output

Examples: narrow-phase collision detection pair testing (only want to report valid collisions), peak detection for template matching.

Expand



- Expand allows element of map operation to insert any number of elements (including none) into its output stream
- Useful when fused with map and other patterns to support variable-rate output

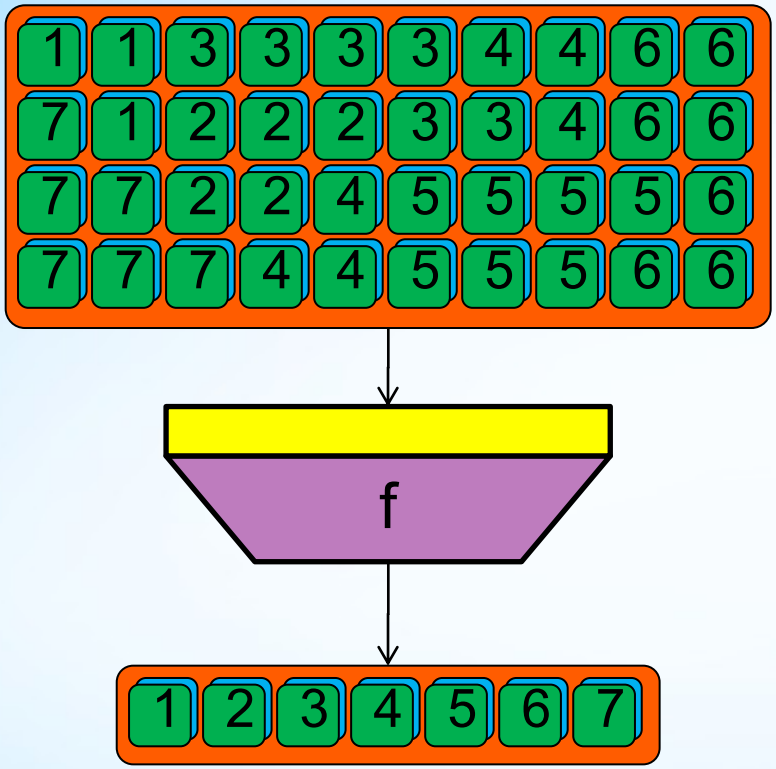
Examples: broad-phase collision detection pair testing (want to report potentially colliding pairs); compression and decompression.

Fused Patterns

- Programs are built from combinations of patterns
- Should be able to fuse patterns for performance
- May be useful to explicitly support specific combinations

- Examples:
  - Gather = map + random read
  - Scatter = map + random write
  - Map + reduce for preprocessing before reduction
  - Map + pack/expand for culling operations
  - Partition + reduce for multidimensional reduction

Search/Match



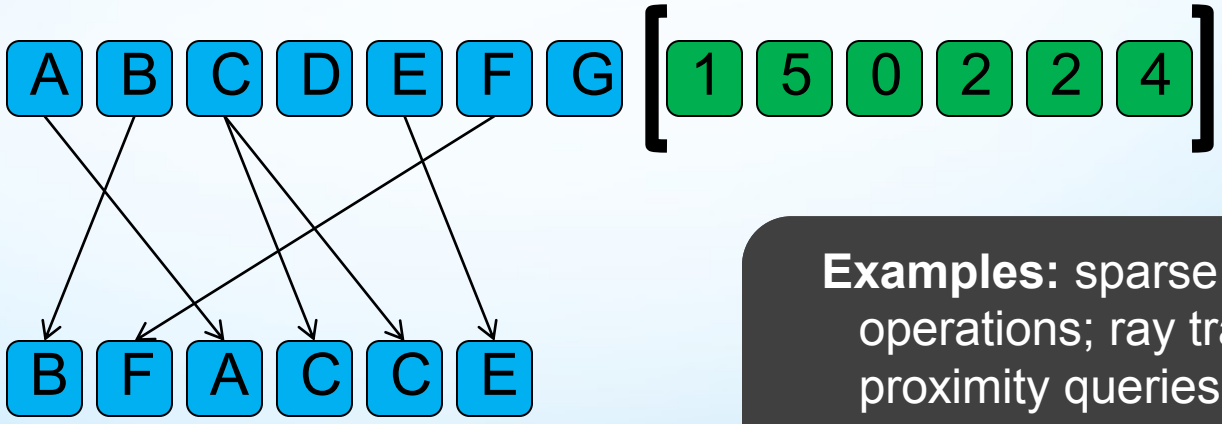
Examples: computation of metrics on segmented regions in vision; computation of web analytics

- Searching and matching fundamental capabilities
- Use to select data for another operation, by creating a (virtual) collection or partitioned collection.

- Example: ***category reduction*** reduces all elements in an array with the same "label", and is the form used in Google's map-reduce

Gather

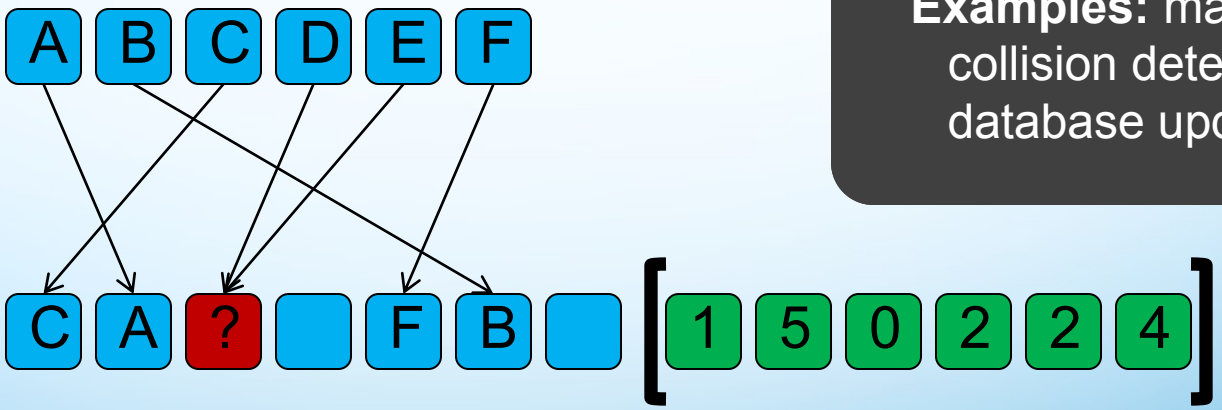
- Map + Random Read
  - Read from a random (computed) location in an array
  - When used inside a map or as a collective, becomes a parallel operation
  - Views into arrays, but no global pointers
  - Write-after-read semantics for kernels to avoid races



Examples: sparse matrix operations; ray tracing; proximity queries; collision detection.

*!Scatter

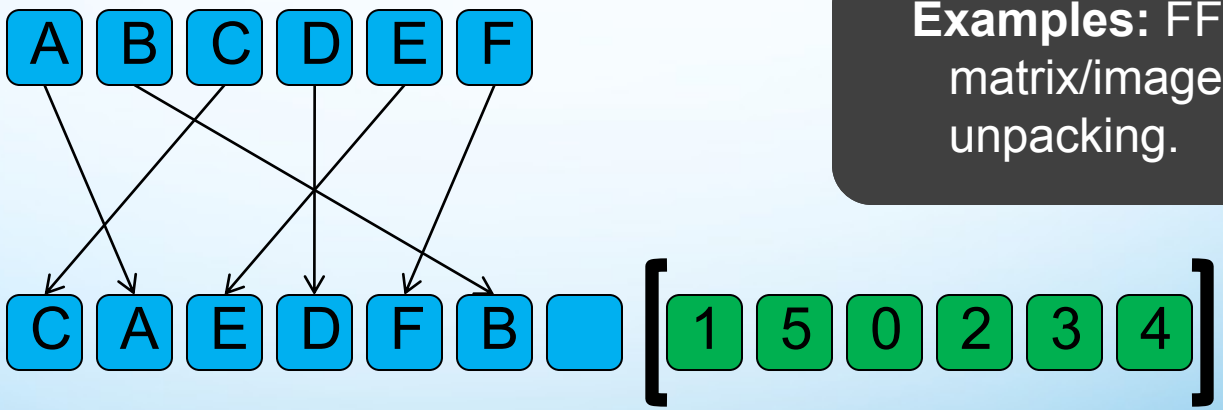
- Map + Random Write
  - Write into a random (computed) location in an array
  - When used inside a map, becomes a parallel operation
  - *Race conditions possible when there are duplicate write addresses ("collisions")*
  - To obtain deterministic scatter, need a deterministic rule to resolve collisions



Examples: marking pairs in collision detection; handling database update transactions.

*Permutation Scatter

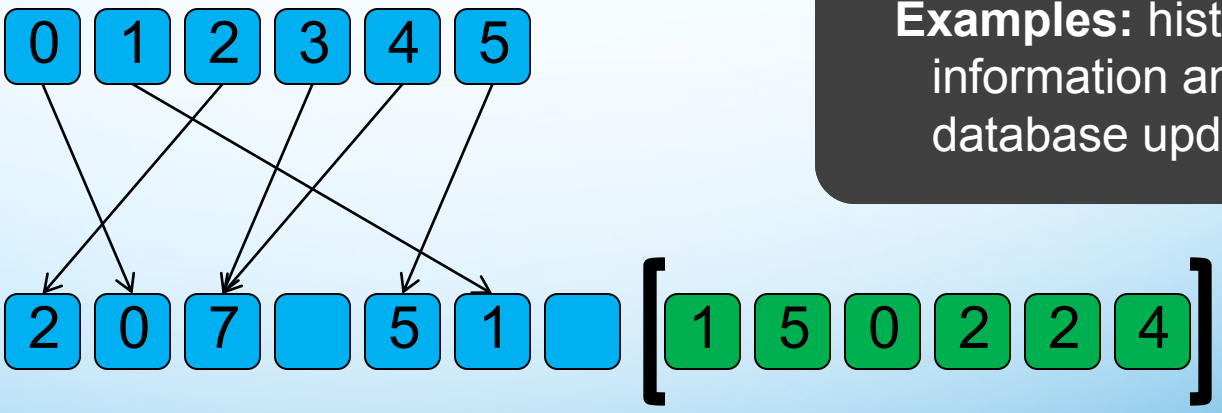
- Make collisions *illegal*
- Only guaranteed to work if no duplicate addresses
- Danger is that programmer will use it when addresses do in fact have collisions, then will depend on undefined behaviour
- Similar safety issue as with out-of-bounds array accesses.
- Can test for collisions in "debug mode"



Examples: FFT scrambling; matrix/image transpose; unpacking.

*Merge Scatter

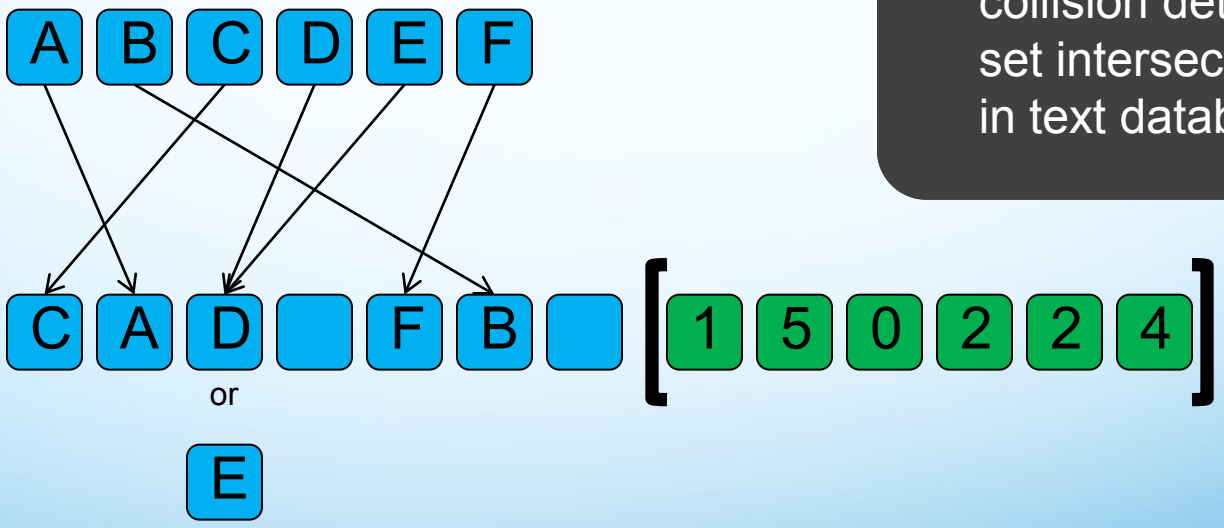
- Use an associative operator to combine values upon collision
- Problem: as with reduce, depends on programmer to define associative operator
- *Gives non-deterministic read-modify-write when used with non-associative operators*
- Due to structured nature of other patterns, can still provide tool to check for race conditions.



Examples: histogram; mutual information and entropy; database updates.

!Atomic Scatter

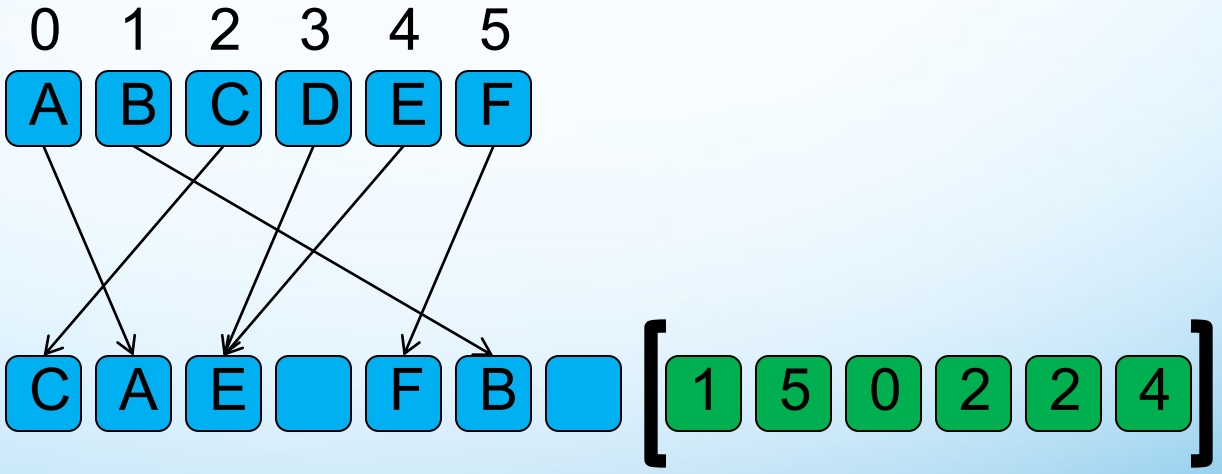
- Resolve collisions atomically but *non-deterministically*
- Use of this pattern will result in non-deterministic programs
- Structured nature of rest of patterns makes it possible to test for race conditions



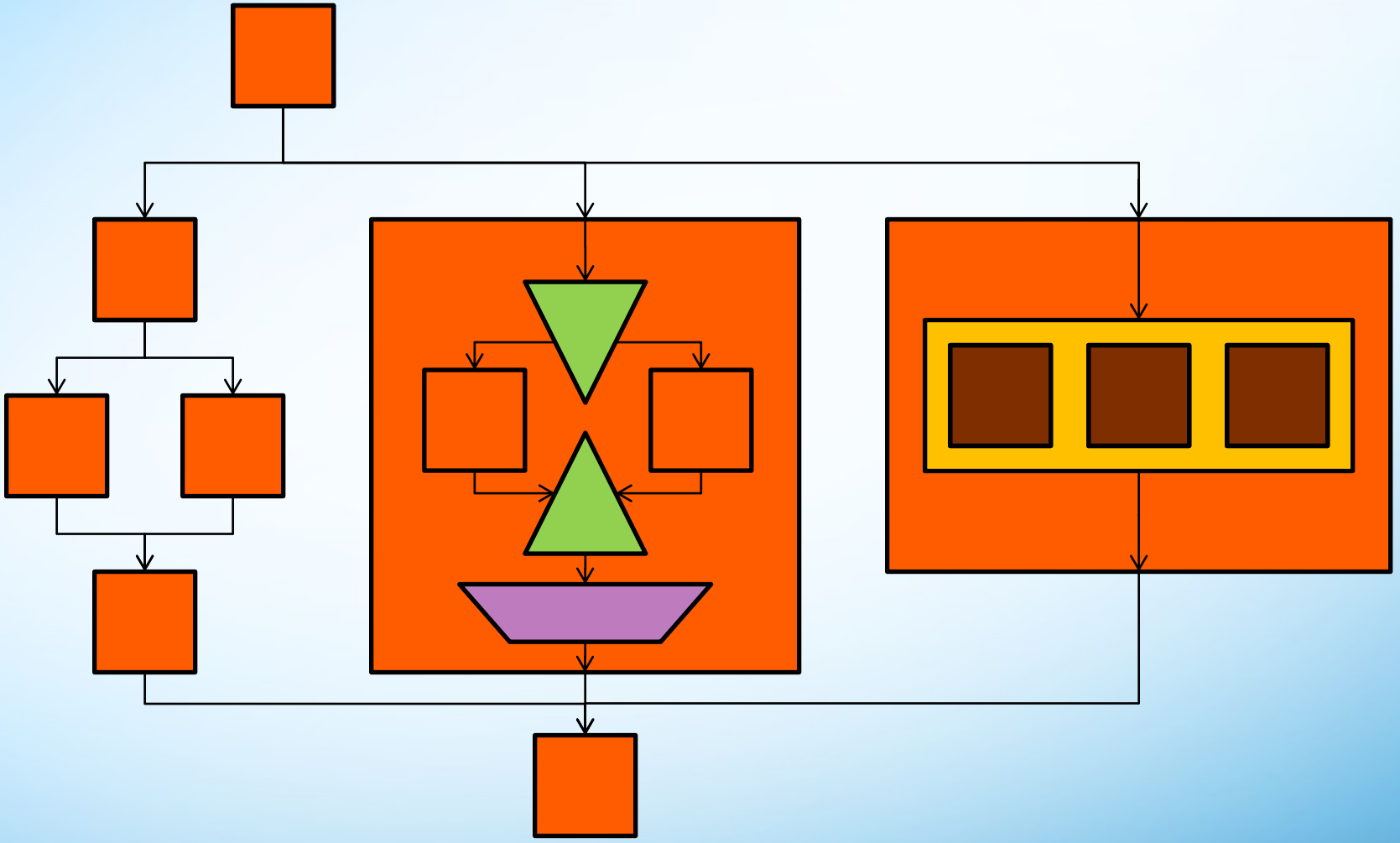
Examples: marking pairs in collision detection; computing set intersection or union (used in text databases)

Priority Scatter

- Assign every parallel element a priority
  - NOTE: Need hierarchical structure of other patterns to do this
- ***Deterministically*** determine "winner" based on priority
- When converting from serial code, priority can be based on original ordering, *giving results consistent with serial program*
- Efficient implementation is similar to hierarchical z-buffer…



Nesting



Conclusion

- Patterns can be used to reason about and organize development of parallel algorithms *and* programming models
  - Integrating these patterns into Ct for heterogeneous computing
- Many useful patterns are deterministic
- Compositions of deterministic patterns lead to deterministic programs

Discussion:

- Are there a smaller number of "primitive" patterns?
- Are any important patterns missing?
- Can "structured" be well-defined?
- How important are non-deterministic patterns?
  - Can any of these be considered "structured"?

BACKUP

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

Intel, Intel Core and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

http://intel.com/software/products

Challenge:
Multiple Parallelism Mechanisms

Modern processors have *many* kinds of parallelism:

- Pipelining
- SIMD within a register (SWAR) vectorization
- Superscalar instruction issue or VLIW
- Overlapping memory access with computation (prefetch)
- Simultaneous multithreading (hyperthreading) per core
- Multiple cores
- Multiple processors
- Asynchronous host and accelerator execution
- HPC adds: clusters, distributed memory, grid…

General Factors Affecting Algorithm Performance

1. **Parallelism**
   - Choose or design a good *parallel* algorithm
   - Large amount of latent parallelism, low serial overhead
   - Asymptotically efficient
   - Should scale to large number of processing elements
2. **Locality**
   - Efficient use of the memory hierarchy
     - More frequent use of faster local memory
   - Coherent use of memory and data transfer
     - Good alignment, predictable memory access; blocking
   - High arithmetic intensity