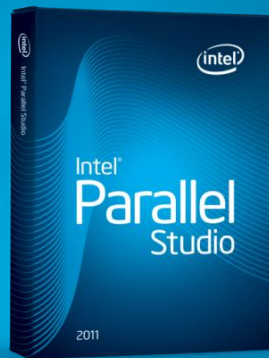
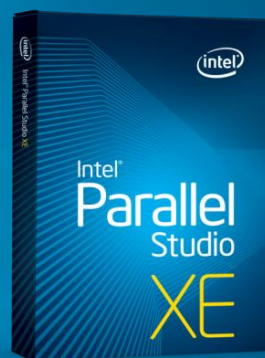


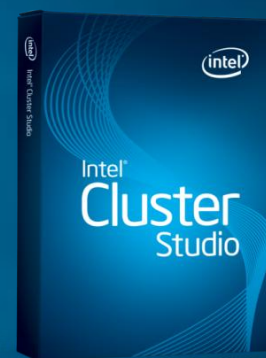
Efficient
Performance



Essential
Performance



Advanced
Performance



Distributed
Performance

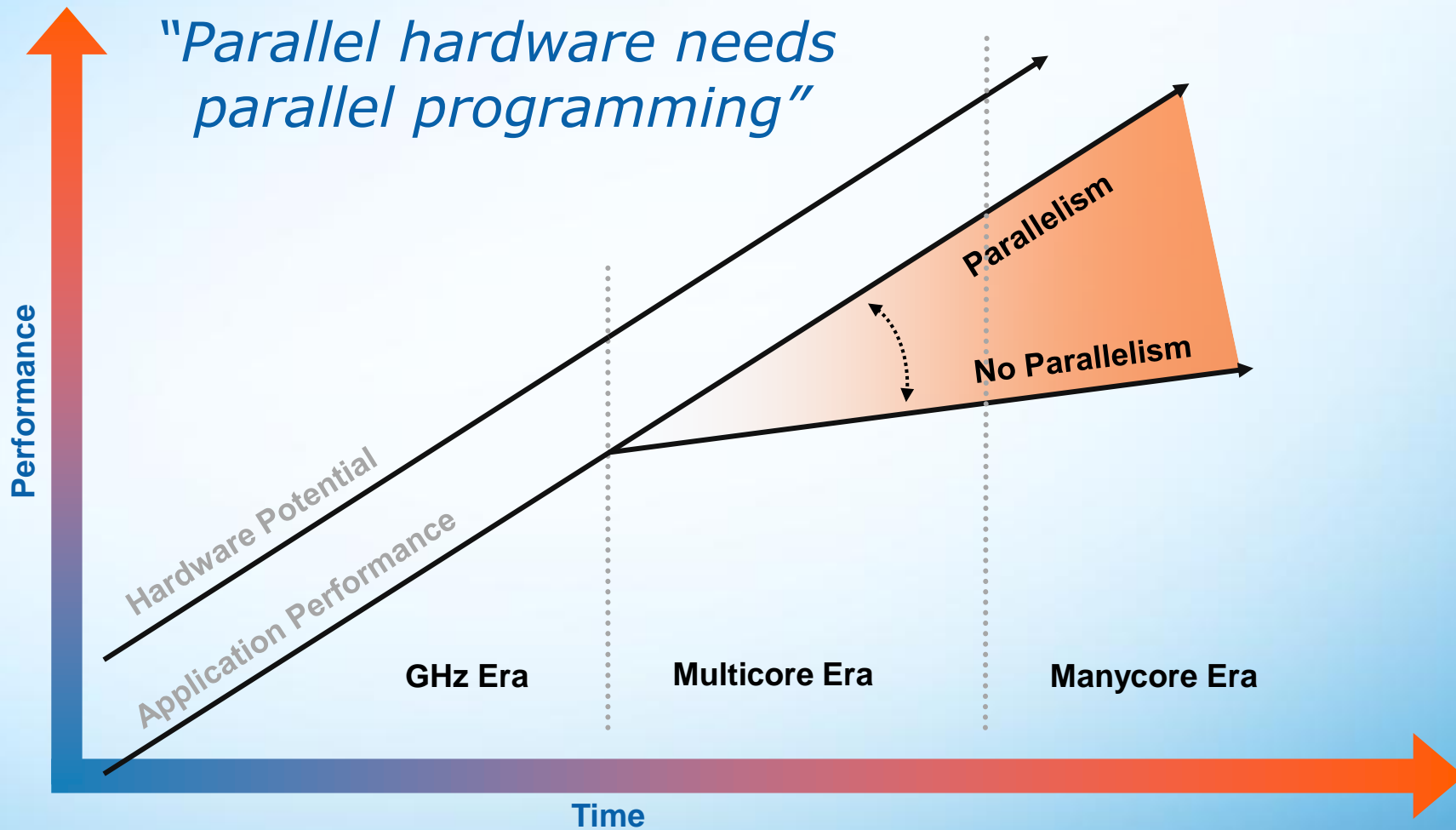
Intel® Parallel Studio XE

Hans Pabst, Developer Product Division

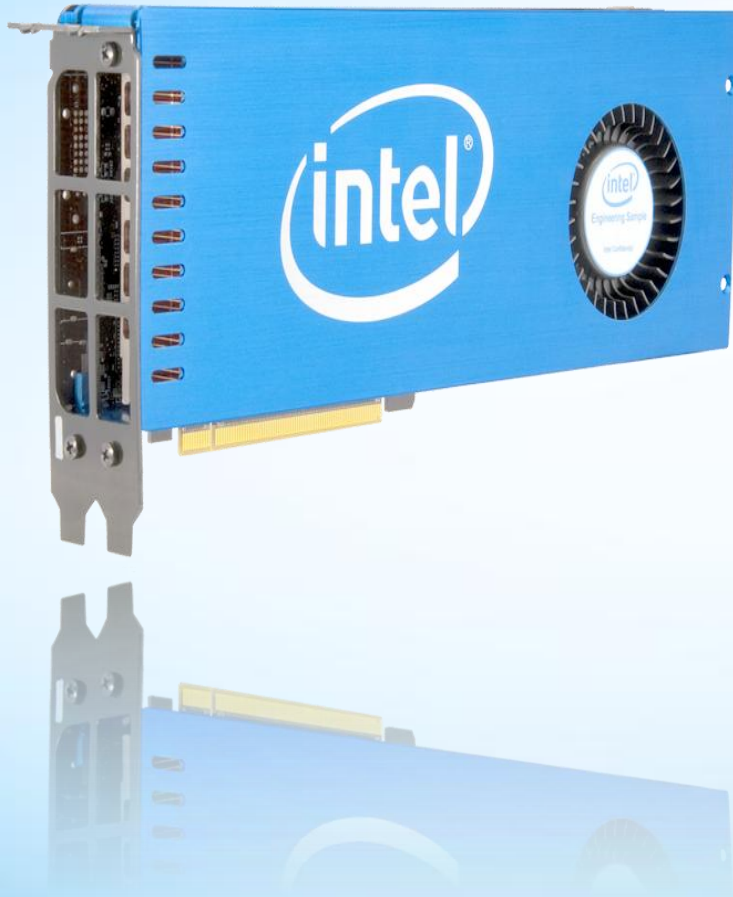
Future Challenges in Tracking and Trigger Concepts

2nd International Workshop, July 8th

Motivation: Performance

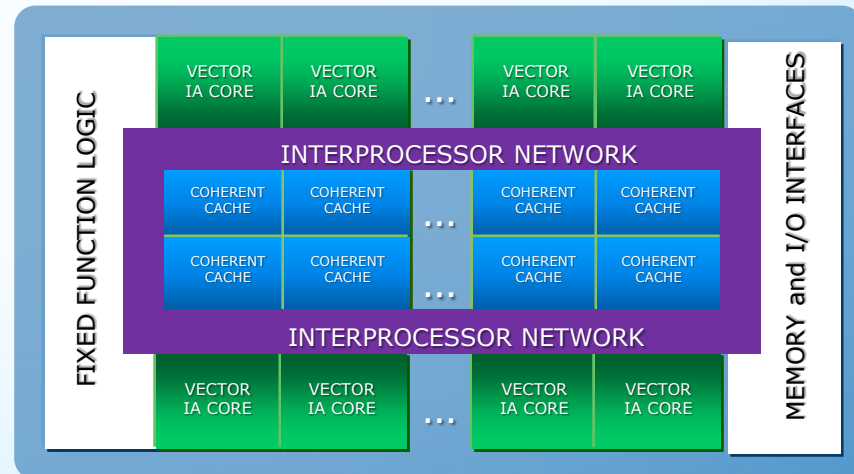


Intel® Many Integrated Core (MIC) Co-Processor Architecture



Knights Ferry Software Development Platform

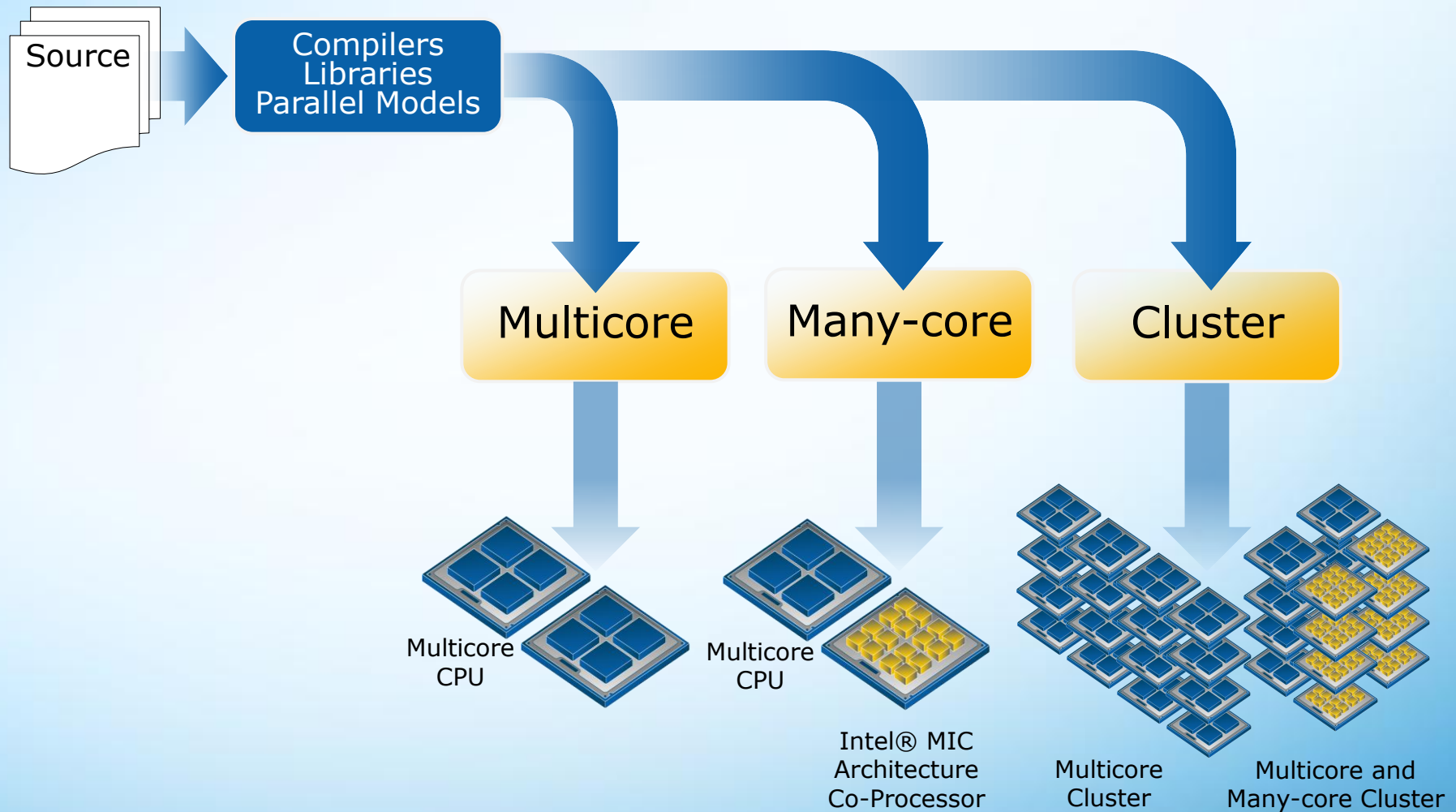
- Up to 32 cores, 128 threads
- 512-bit SIMD support
- Fully coherent cache
- Up to 2 GB GDDR5 memory
- Latest Intel SW developer products



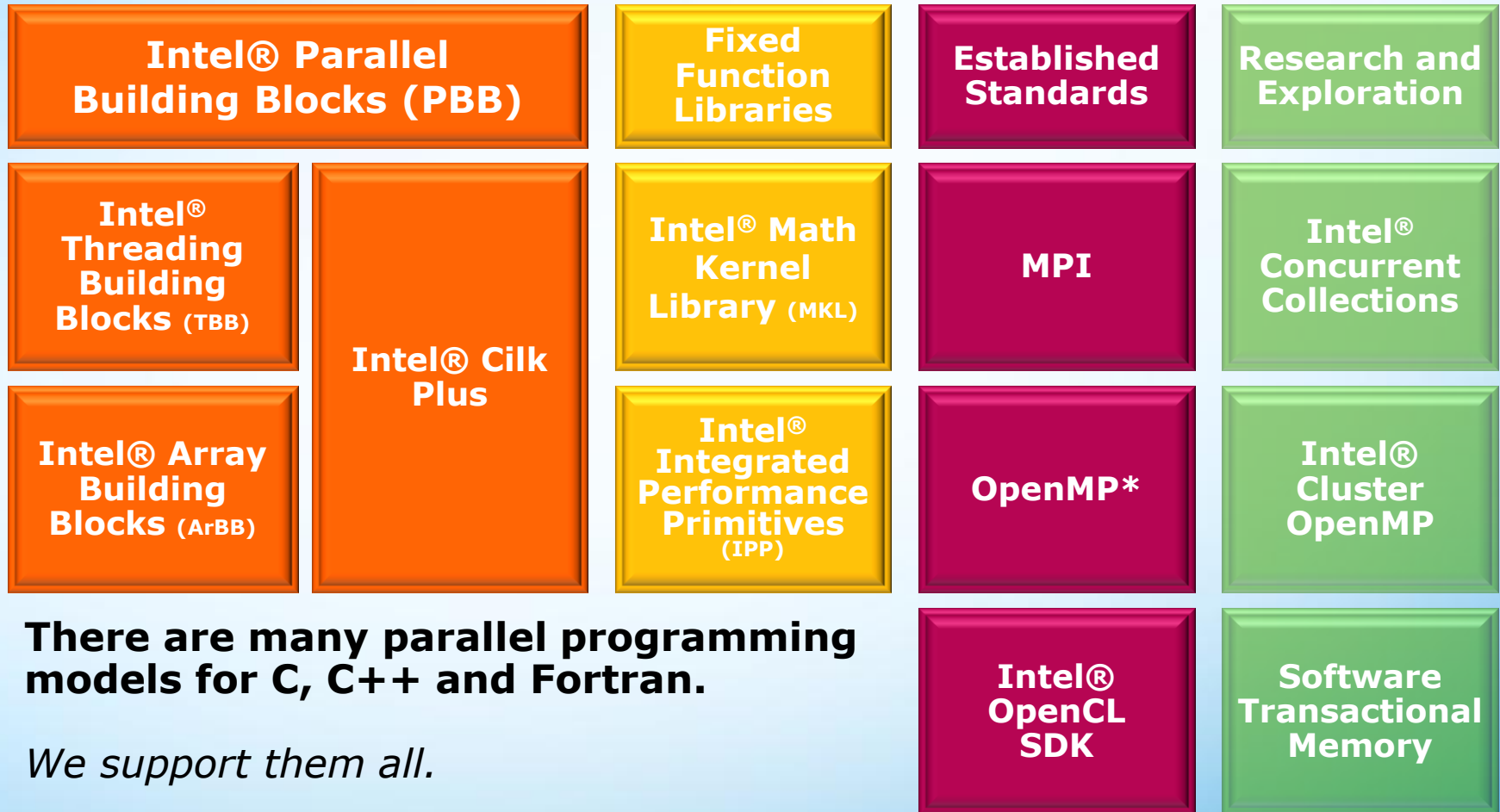
First Intel® MIC product

- Codenamed "Knights Corner"
- Planned for production on Intel's 22 nm 3D Tri-Gate transistor technology

One source base, tuned to many targets



Programming Models and Libraries



There are many parallel programming models for C, C++ and Fortran.

We support them all.

Performance Optimization Steps

Shown steps enable to scale forward to many-core co-processors.

Baseline
Recompilation of the existing code.

Intel® Compiler
- Performance comparison with other compilers.

Intel® Libraries
Identify fixed functionality and employ optimized code, threads, and (with Intel® MKL) multiple nodes.

Intel® IPP
- Multi-media
- etc.

Intel® MKL
- Statistics (VSL)
- BLAS
- etc.

Multithreading
Achieve scalability across multiple cores, sockets, and nodes.

Intel® Compiler
- Auto/guided par.
- OpenMP*

Intel® Parallel Building Blocks
- Intel TBB
- Intel Cilk Plus
- Intel ArBB

Intel® Cluster Studio
- Cluster tools
- MPI

Vectorization
Make use of SIMD extensions, e.g. Intel® AVX.




Intel® Compiler
- Optimization hints
- #pragma simd

Intel® Cilk Plus
- Array notation
- Elemental fn.

Intel® ArBB
- Unified model for SIMD and threads

The order of the steps is suggested to be based on a performance analysis.

Intel® Parallel Studio XE

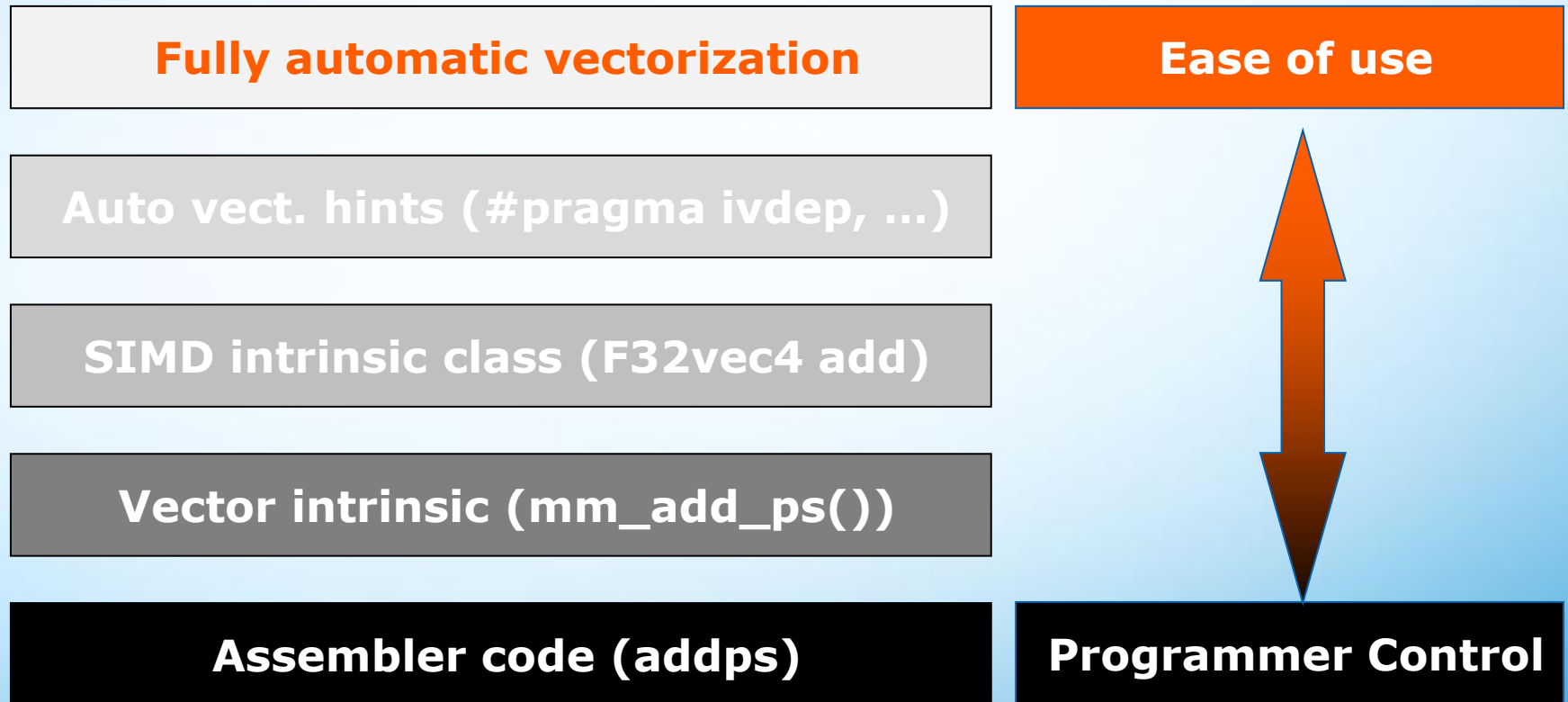
Phase	Tool	Usage	Benefit
Compilation Debugging	 Intel® Composer XE	C/C++ and Fortran Compiler, Performance Libraries and parallel programming models	Strong step towards higher performance (ad- hoc and in the future), additional robustness and safety
Verification Correctness	 Intel® Inspector XE	Debugging (memory access and thread usage) for better code / application quality	Higher productivity, early or continuous quality assurance (GUI+CLI)
Analysis Tuning	 Intel® VTune™ Amplifier XE	Profiler to inspect hardware events (counter), scalability etc.	Avoids work based on guesses, combines ease of use with deep insight (GUI+CLI)

Powerful compilers. Verification and performance analysis tools supporting continuous integration.

Intel® Compiler

Intel® C/C++ Compiler Version 12
Intel® Fortran Compiler Version 12

SIMD Vectorization



SIMD Vectorization

Intel® Parallel Building Blocks

Fully automatic vectorization

Ease of use

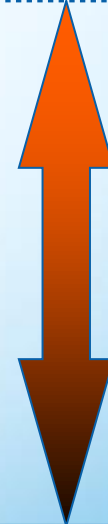
Auto vect. hints (#pragma ivdep, ...)

SIMD intrinsic class (F32vec4 add)

Vector intrinsic (mm_add_ps())

Assembler code (addps)

Programmer Control



Basic Vectorization – Switches [1]

{L&M} **-x<extension>** {W}: **/Qx<extension>**

- Targeting Intel® processors - specific optimizations for Intel® processors
- Compiler will try to make use of all instruction set extensions up to and including <extension>; for Intel® processors only!
- Processor-check added to main-program
- Application will not start (will display message), in case feature is not available

{L&M}: **-m<extension>** {W}: **/arch:<extension>**

- No Intel processor check; does not perform Intel-specific optimizations
- Application is optimized for and will run on both Intel and non-Intel processors
- Missing check can cause application to fail in case extension not available

{L&M}: **-ax<extension>** {W}: **/Qax<extension>**

- Multiple code paths – a 'baseline' and 'optimized, processor-specific' path(s)
- Optimized code path for Intel® processors defined by <extension>
- Baseline code path defaults to -msse2 (Windows: /arch:sse2); can be modified by -m or -x (/Qx or /arch) switches

Basic Vectorization – Switches [2]

The default is `-msse2` (Windows: `/arch:sse2`)

- Activated implicitly for `-O2` or higher
- Implies the need for a target processor with Intel® SSE2
- Use `-mia32` (`/arch:ia32`) for 32-bit processors without SSE2 (e.g. Intel® Pentium™ 3) to adjust baseline code path

Special switch `-xHost` (Windows: `/QxHost`)

- Compiler checks host processor and makes use of latest instruction set extension available
- Avoid for builds being executed on multiple, unknown platforms

Multiple extensions can be used in combination:

`-ax<ext1>,<ext2>` (Windows: `/Qax<ext1>,<ext2>`)

- Can result in more than 2 code paths (incl. baseline code path)
- Use `-mia32` (`/arch:ia32`) for 32-bit processors without SSE2 (e.g. Intel® Pentium™ 3) to adjust baseline code path

Vectorization – More Switches/Directives

Disable vectorization

- Globally via switch: {L&M}: **-no-vec** {W}: **/Qvec-**
- For a single loop: directive **novector**
 - Disabling vectorization here means not using packed SSE/AVX instructions. The compiler still might make use of the corresponding instruction set extensions.

Enforcing vectorization for a loop (overwrite compiler heuristics)

#pragma vector always

- will enforce vectorization even if the compiler thinks it is not profitable to do so (e.g due to non-unit strides or alignment issues)
- Will not enforce vectorization if the compiler fails to recognize this as a semantically correct transformation
- Using directive **#pragma vector always assert** will print error message in case the loop cannot be vectorized and will abort compilation

Vectorization Report

- Provides details on vectorization success & failure
 - L&M: `-vec-report<n>`, n=0,1,2,3,4,5
 - W: `/Qvec-report<n>`, n=0,1,2,3,4,5

```
35:  subroutine fd( y )
36:  integer :: i
37:  real, dimension(10), intent(inout) :: y
38:  do i=2,10
39:      y(i) = y(i-1) + 1
40:  end do
41:  end subroutine fd
```

```
novec.f90(38): (col. 3) remark: loop was not vectorized: existence of
vector dependence.
novec.f90(39): (col. 5) remark: vector dependence: proven FLOW
dependence between y line 39, and y line 39.
novec.f90(38:3-38:3):VEC:MAIN_: loop was not vectorized: existence of
vector dependence
```


Diagnostic Level of Vectorization Switch

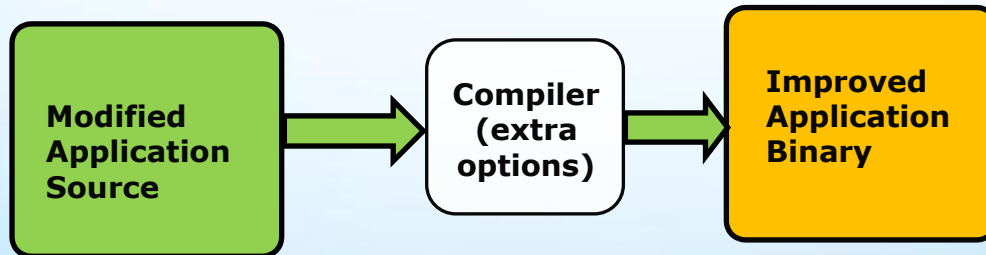
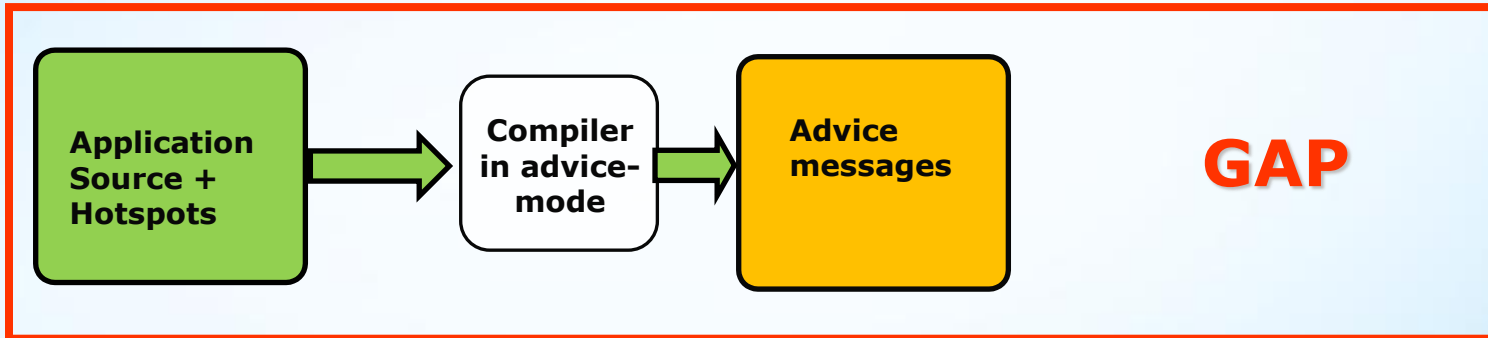
L&M: -vec-report<N> W: /Qvec-report<N>

N	Diagnostic Messages
0	No diagnostic messages; same as not using switch and thus default
1	Report about vectorized loops– default if switch is used but N is missing
2	Report about vectorized loops and non-vectorized loops
3	Same as N=2 but add add information on assumed and proven dependencies
4	Report about non-vectorized loops
5	Same as N=4 but add detail on why vectorization failed

Note:

- In case inter-procedural optimization (-ipo or /Qipo) is activated and compilation and linking are separate compiler invocations, the switch needs to be added to the link step

Compiler as a Tool



Simplifies programmer effort in application tuning

Vectorization Example

```
void mul(NetEnv* ne, Vector*
    rslt
    Vector* den, Vector* flux1,
    Vector* flux2, Vector* num
{
    float *r, *d, *n, *s1, *s2;
    int i;
    r=rslt->data;
    d=den->data;
    n=num->data;
    s1=flux1->data;
    s2=flux2->data;

    for (i = 0; i < ne->len; ++i)
        r[i] = s1[i]*s2[i] +
            n[i]*d[i];
}
```

GAP Messages (simplified):

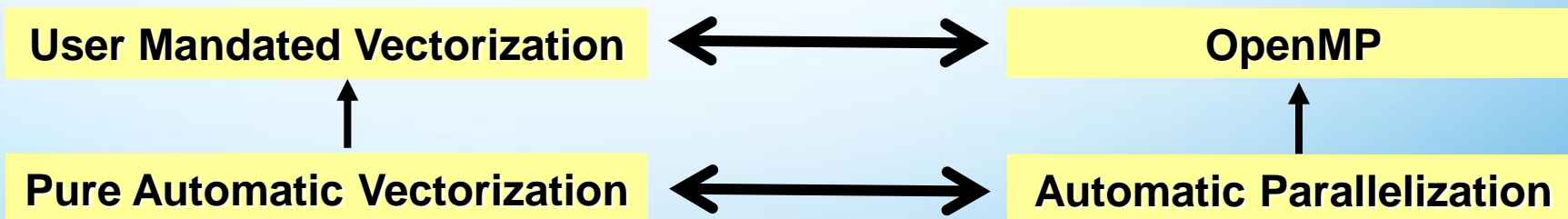
1. Use a local variable to store the upper-bound of loop at line 29 (variable: `ne->len`) if the upper-bound does not change during execution of the loop
 2. Use “#pragma ivdep” to help vectorize the loop at line 29, if these arrays in the loop do not have cross-iteration dependencies: `r, s1, s2, n, d`
- > Upon recompilation, the loop will be vectorized

User-Mandated Vectorization

User-mandated vectorization is based on a new **SIMD Directive**

- The SIMD directive provides additional information to compiler to enable vectorization of loops (at this time only inner loop)
- Supplements automatic vectorization but differently to what traditional directives like IVDEP, VECTOR ALWAYS do, the SIMD directive is more a command than a hint or an assertion: The compiler heuristics are completely overwritten as long as a clear logical fault is not being introduced

Relationship similar to OpenMP versus automatic parallelization:



SIMD Directive Notation

C/C++: **#pragma simd [clause [,clause] ...]**

Fortran: **!DIR\$ SIMD [clause [,clause] ...]**

Without any additional clause, the directive enforces vectorization of the (innermost) loop

Example:

```
void add_f1(float* a, float* b, float* c, float* d, float* e, int n)
{
    #pragma simd
    for (int i=0; i<n; i++)
        a[i] = a[i] + b[i] + c[i] + d[i] + e[i];
}
```

Without the SIMD directive, vectorization will fail (too many pointer references to do a run-time overlap-check).

Intel® Parallel Building Blocks

Intel Cilk Plus
Intel TBB
Intel ArBB

Intel® Parallel Building Blocks (Intel PBB)

- Programming models in Intel PBB
 - **Composable**, choice to mix and match
 - **Portable**, and **performance portable**
 - **Productive**, and **safe**

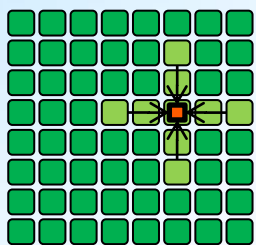
- Parallel patterns

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

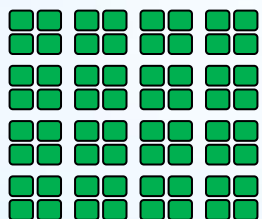
1. A **small number of patterns** can support a **wide range of applications**.
2. Supporting “good” patterns directly leads to **higher productivity**.
3. In addition, a useful subset of “good” patterns are structured and deterministic.

Parallel Patterns

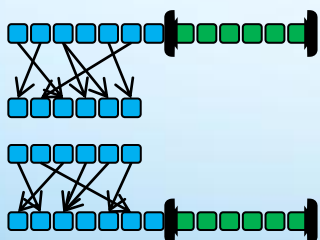
- Stencil



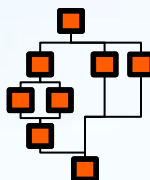
- Partition



- Gather/scatter



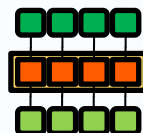
- Superscalar sequence



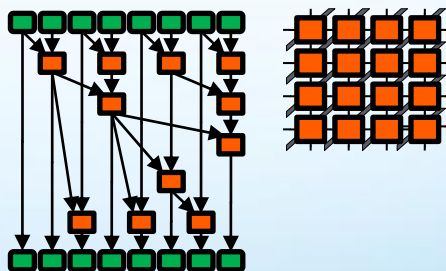
- Speculative selection



- Map



- Scan and Recurrence



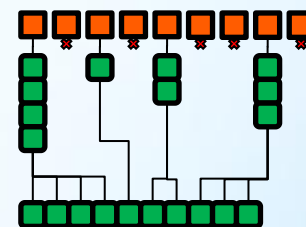
- Pipeline



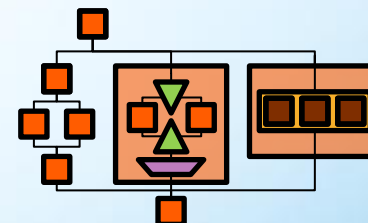
- Reduce



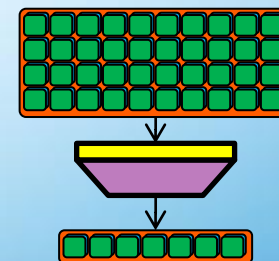
- Pack & Expand



- Nest



- Search & Match



Parallel Patterns in Intel PBB

- **Cilk Plus**

- `cilk_spawn`: nesting (fork-join)
- Hyperobjects: reduction
- `cilk_for`, elemental functions: map
- Array notation: scatter, gather

- **Threading Building Blocks**

- `parallel_invoke`, task-graph: nesting (fork-join)
- `parallel_for`, `parallel_foreach`: map
- `parallel_do`: workpile (map + incr. task addition)
- `parallel_reduce`, `parallel_scan`: reduce, scan
- `parallel_pipeline`: pipeline

- **Array Building Blocks**

- Elemental functions: map
- Collective operations: reduce, scan
- Permutation operations: pack, scatter, gather

Intel® Cilk™ Plus

Elemental Functions and Array Sections

Intel Cilk Plus, est. 2010

Elemental Functions and Array Sections

- Elemental functions (“kernels”)
 - Properties apply to guarantee vectorization
- Array notation (sections/slices)
 - [start:size], or
 - [start:size:stride]

Example: Element-wise Addition

```
__declspec(vector) void kernel(int& result, int a, int b)  
{  
    result = a + b;  
}
```

```
void sum(const int* begin, const int* begin2,  
        std::size_t size, int* out)  
{  
    cilk_for (std::size_t i = 0; i < size; ++i) {  
        kernel(out[i], begin[i], begin2[i]);  
    }  
}
```

```
void sum2(const int* begin, const int* begin2,  
         std::size_t size, int* out)  
{  
    kernel(out[0:size], begin[0:size], begin2[0:size]);  
}
```

Intel® Array Building Blocks

Intel ArBB

ArBB Overview

C++ Library

- Embedded Language
- Dynamic Compiler

Vector-parallel

- Containers and parallel operations
- Implicit use of SIMD and threads

Scalable

- Scalable across diff. SIMD widths
- Scalable across multiple cores

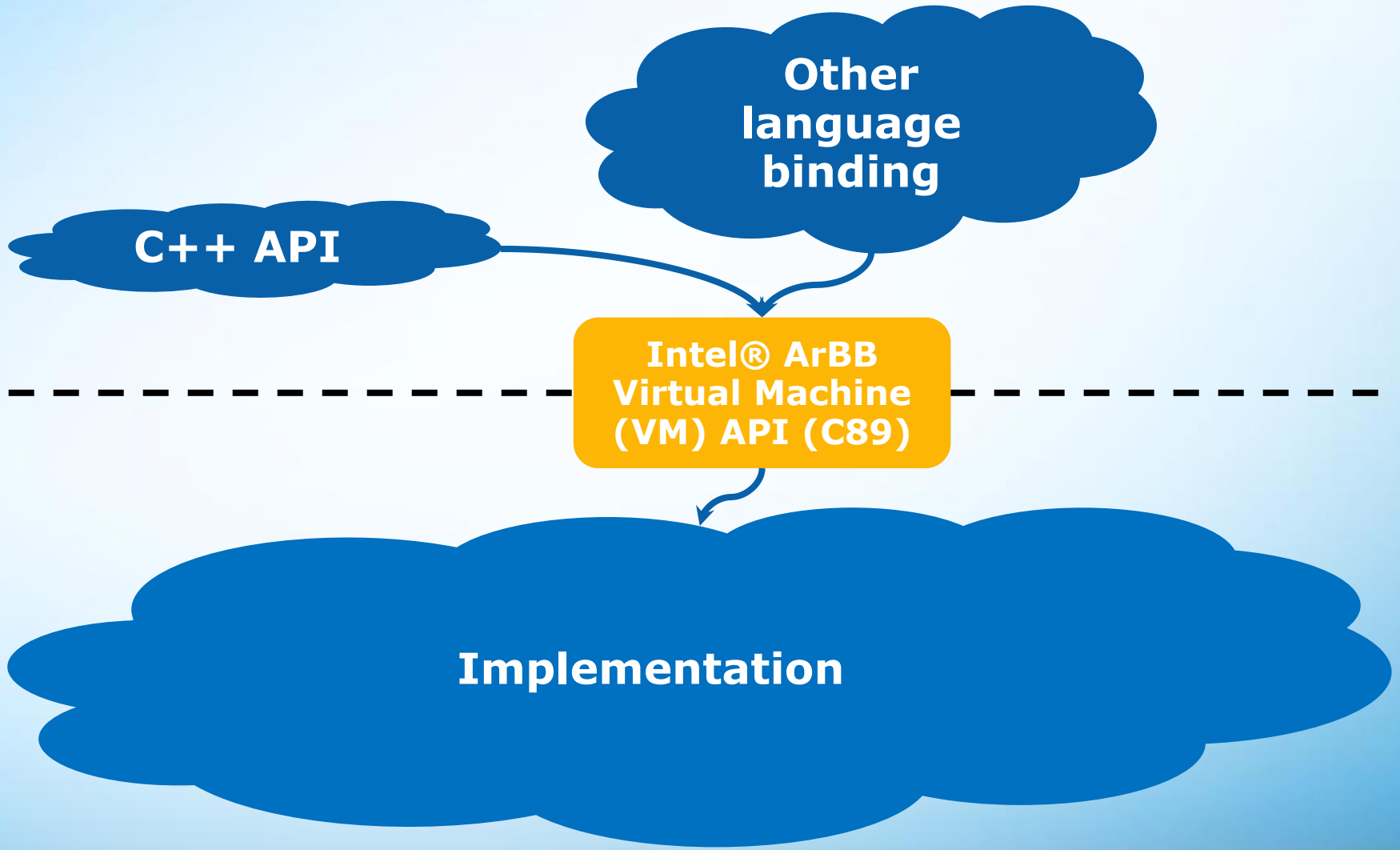
Deterministic

- Independent of #cores utilized
- No sync. objects, no data races

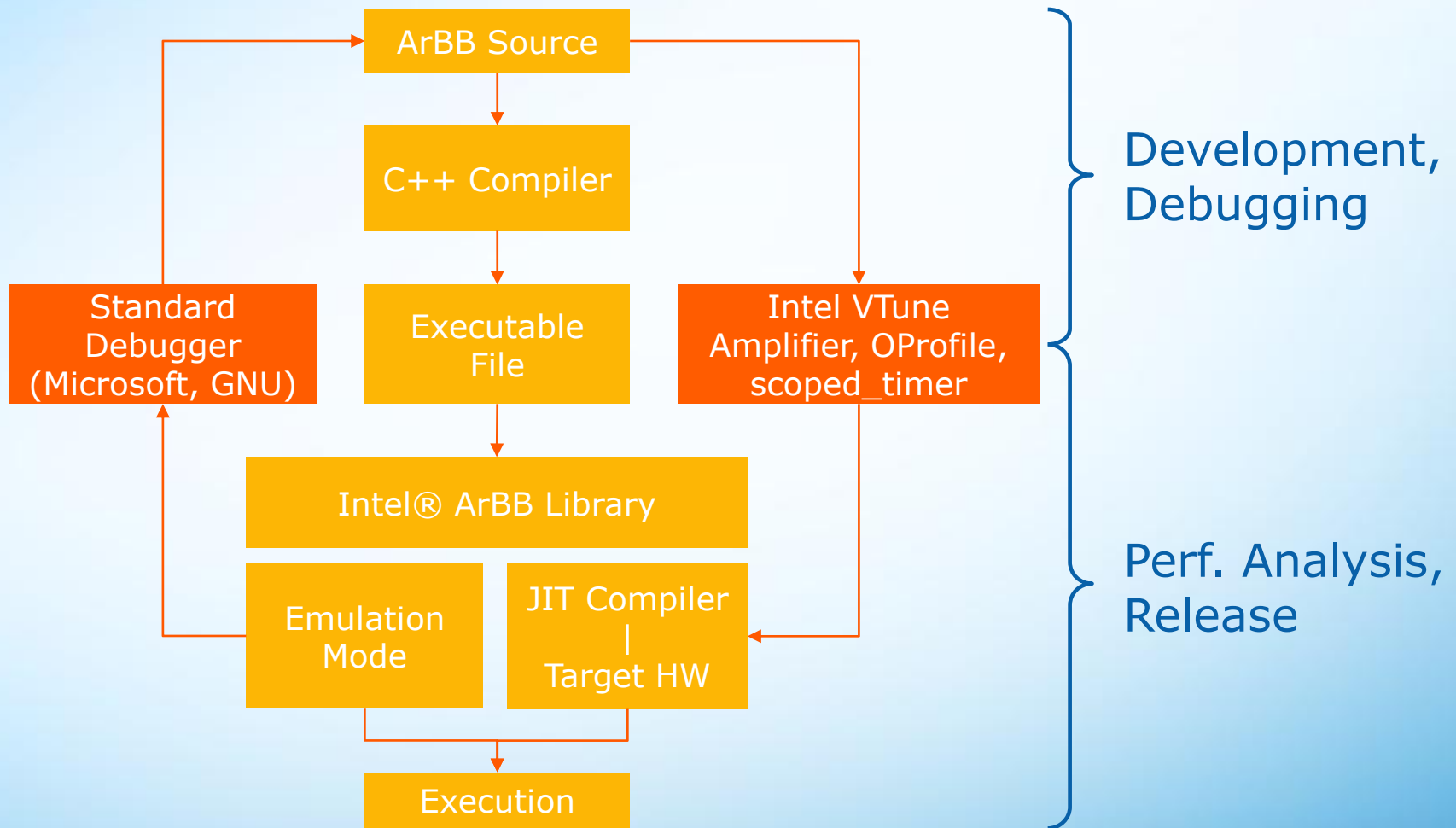
Safe

- Separate memory space (data)
- No pointers, by-value semantic

Programming Interfaces



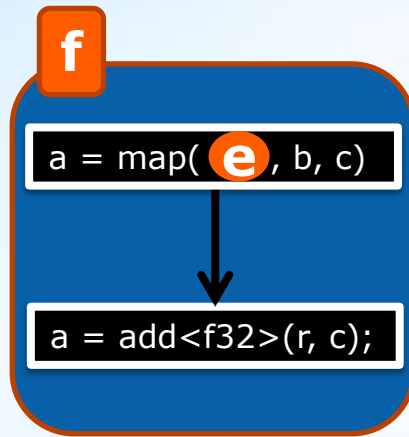
Development Cycle and Tools



Roadmap: "emulation mode" and native code execution will be aligned (same precision, debugging native code).

Vector Processing and Elemental Functions

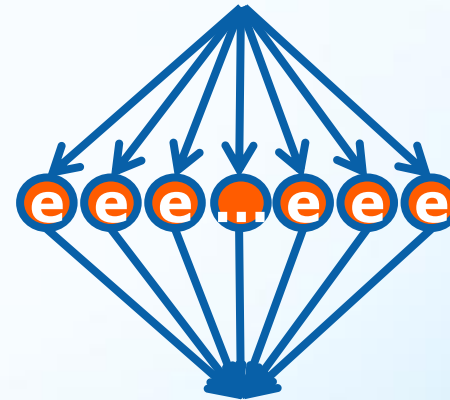
Vector Processing



```
void f(dense<f32>& result,  
      dense<f32> a, dense<f32> b)  
{  
    result = a * b;  
}
```

```
dense<f32> result, a, b;  
call(f)(result, a, b);
```

Elemental Processing



```
void e(f32& result, f32 a, f32 b)  
{  
    result = a * b;  
}
```

"Kernel"

Elemental processing is naturally embedded into a more general vector processing context allowing e.g. scatter etc.

ArBB Function?

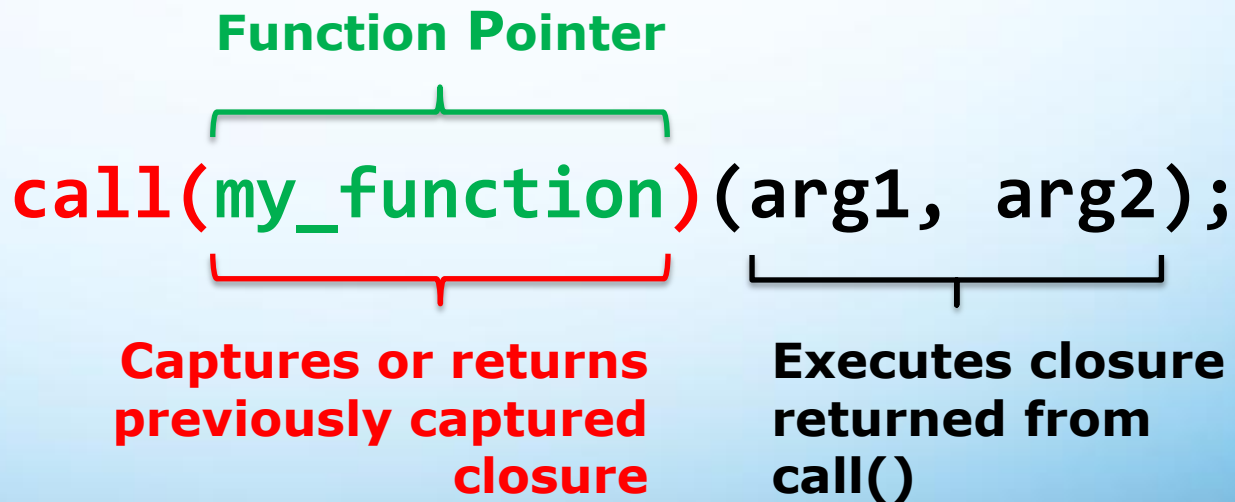
void function(*signature*)

- Void Multiple results possible
 - Via argument list
- Signature **Outputs/in-outs**
 - By "non-const reference"**Inputs**
 - By "const reference", or
 - By-value
- Types Intel ArBB Types

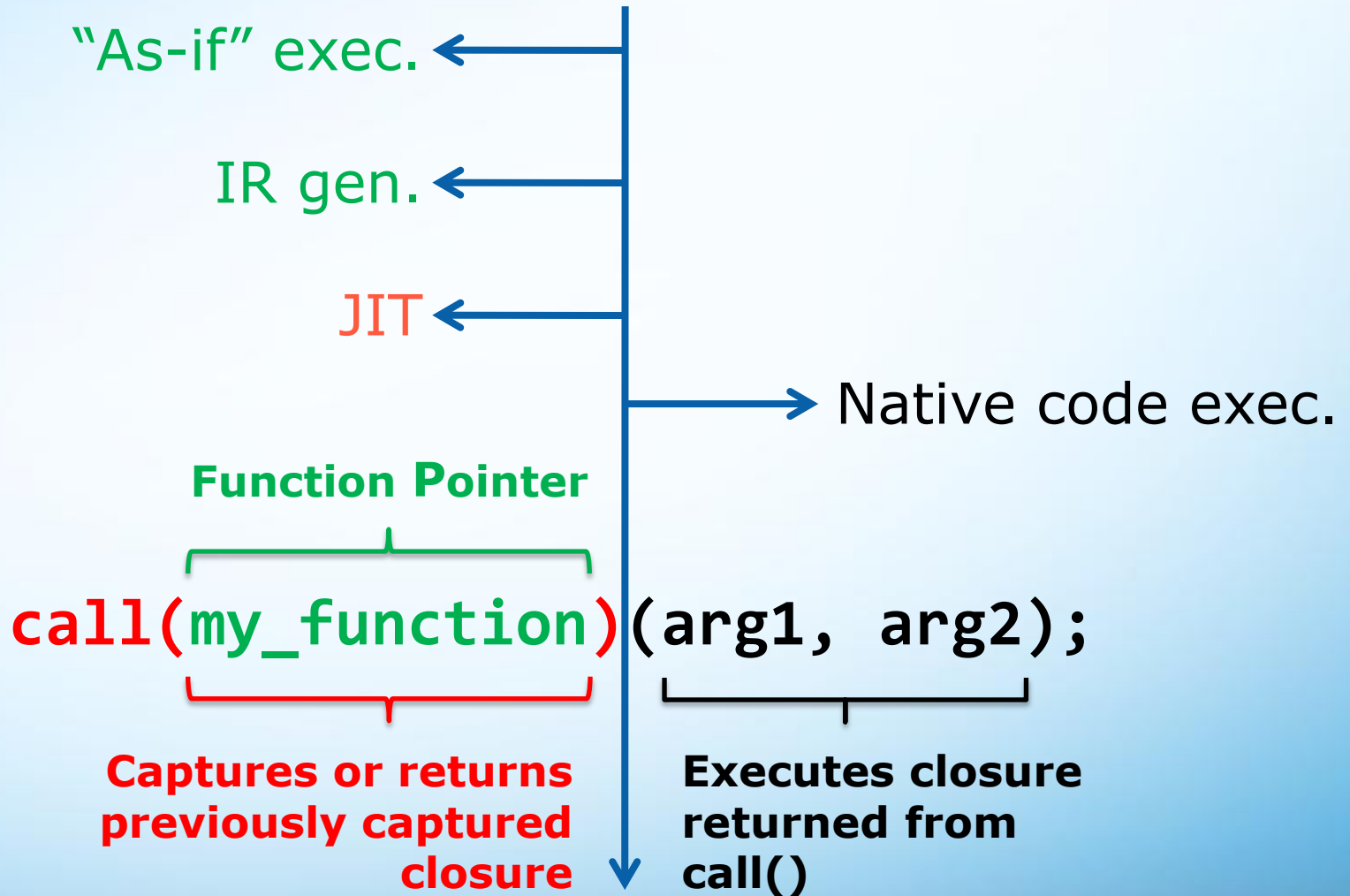
arbb::call()

In a narrow sense an "ArBB function" does not exist since it is regular C++ code.

JIT Compiler and Code Generation



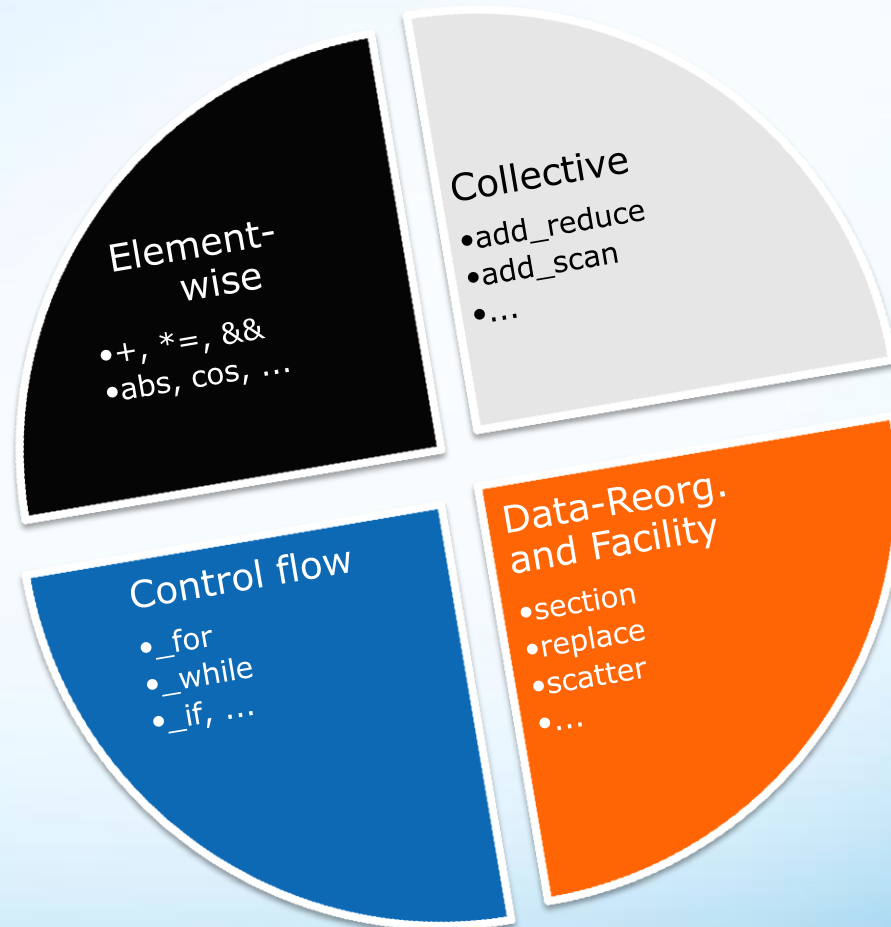
JIT Compiler and Code Generation



Scalar Types

Type	Description	C++
f32, f64	32/64 bit floating point	float, double
i8, i16, i32, i64	8/16/32 bit integer (signed)	signed char, short, int
u8, u16, u32, u64	8/16/32 bit integer (unsigned)	unsigned char, unsigned short, unsigned int
boolean	Boolean value (true/false)	bool
usize, isize	Index type	size_t (ssize_t)

Operations

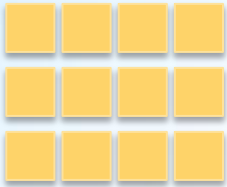


Data Management



// one-dimensional array with double-precision numbers

```
dense<f64> signal (  
    "{ 0.3, 5.3, -2.4, 3.1 }");
```



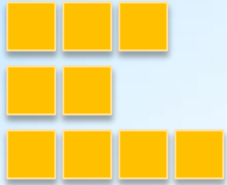
// two-dimensional array with signed integers (byte)

```
dense<i8,2> greyscale_image (  
    "{ { 0, 12, 34, 21 }, { 45, 31, 21, -74 },  
      { 81, 52, 22, 15 } }");
```

Segregated storage and memory management

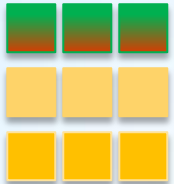
- Transparent on remote execution
- No pointers, logically “by value”

Data Management



// irregular array of signed integers

```
nested<i32> graph_edges (  
    "{ { 0, 1, 3 }, { 3, 5 }, { 1, 2, 8, 9 } }");
```



// one-dimensional array of structured type (user-defined)

```
struct particle { f64 m; i8 f; i32 c; };  
dense<particle> state (  
    "{ { 0.3, 1, 5 }, { 0.5, 2, 2096 },  
    { 3.2, 56, 7 } }");
```

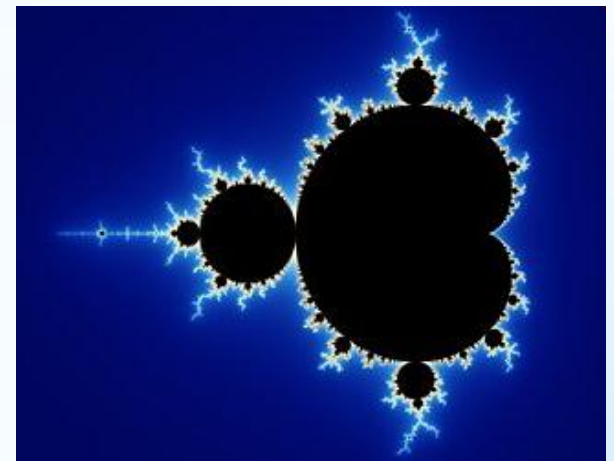
- Type of container elements
 - Scalar type (Intel ArBB)
 - Collection of scalar types (structured, heterogeneous)
 - Nested structure of the above types
- “Arrays of Structures” (AoS) are stored as “Structures of Arrays” (SoA) internally

Example: Mandelbrot

```
int max_count = 4711;
void mandel(i32& d, std::complex<f32> c) {
    i32 i;
    std::complex<f32> z = 0.0f;
    _for (i = 0, i < max_count, i++) {
        _if (abs(z) >= 2.0f) {
            _break;
        } _end_if;
        z = z * z + c;
    } _end_for;
    d = i;
}

void doit(dense<i32,2>& d, const dense<std::complex<f32>,2>& c)
{
    map(mandel)(d, c);
}

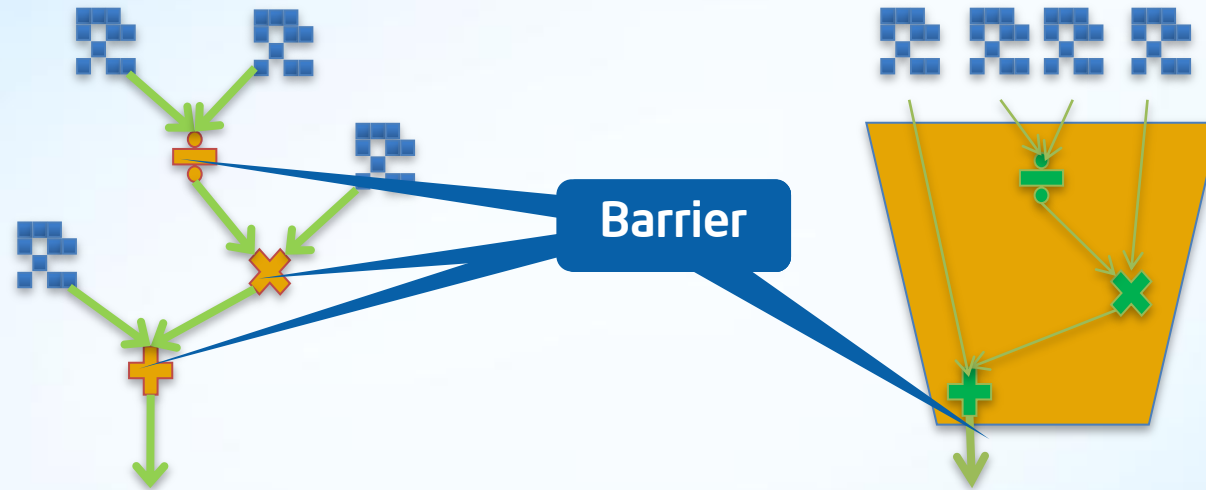
bind(pos, c_pos, cols, rows);
bind(dest, c_dest, cols, rows);
call(doit)(dest, pos);
```



Color Legend:

- ArBB State (Type)
- ArBB Behavior

High-Level Optimizations



- Intel® ArBB operators: separately written, but fused code across call boundaries (inlining, despite of modularization)
 - Higher arithmetic intensity per task
 - Higher memory locality
 - Less scheduling overhead
 - Less synchronization

Example: Matrix-Vector Multiplication

How to do it in C++?

```
for (int j = 0; j < n; ++j) {  
    result[j] = matrix[j * n] * vector[0];  
    for (int i = 1; i < m; ++i) {  
        result[j] += matrix[j * n + i]  
                    * vector[i];  
    }  
}
```

Example: Matrix-Vector Multiplication

*It is often possible to eliminate loops entirely.
Express what to do, instead of how to do it.*

```
usize nrows = matrix.num_rows();  
result = add_reduce(matrix  
    * repeat_row(vector, nrows));
```

Example: Matrix-Vector Multiplication

... But loops are just fine in cases where the loop body contains enough parallel work

```
_for (usize i = 0, i < nrows, ++i) {  
    result[i] = add_reduce(matrix.row(i)  
        * vector);  
} _end_for;
```


Example: Query Value Locations

```
void findval(dense<usize>& result,  
            const dense<f32> in,  
            f32 value)  
{  
    dense<boolean> hits = in == value;  
    dense<usize> pos = indices(0, in.length());  
    result = pack(pos, hits);  
}
```

Intel® Array Building Blocks

- Download Intel ArBB and try it!

<http://intel.com/go/arbb/>

- Documentation, articles, and user forum

<http://software.intel.com/en-us/articles/intel-array-building-blocks-documentation/>

<http://software.intel.com/en-us/articles/intel-array-building-blocks-kb/all/>

<http://software.intel.com/en-us/forums/intel-array-building-blocks/>

Questions?



Optimization Notice

Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20110307

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.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Cilk, Core Inside, FlashFile, i960, InstantIP, Intel, the Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.

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

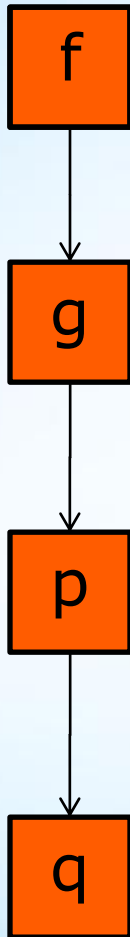
Copyright © 2011. Intel Corporation.

<http://intel.com/software/products>

Parallel Patterns

Backup Slides

(Serial) Sequence



A serial sequence is executed in the exact order given:

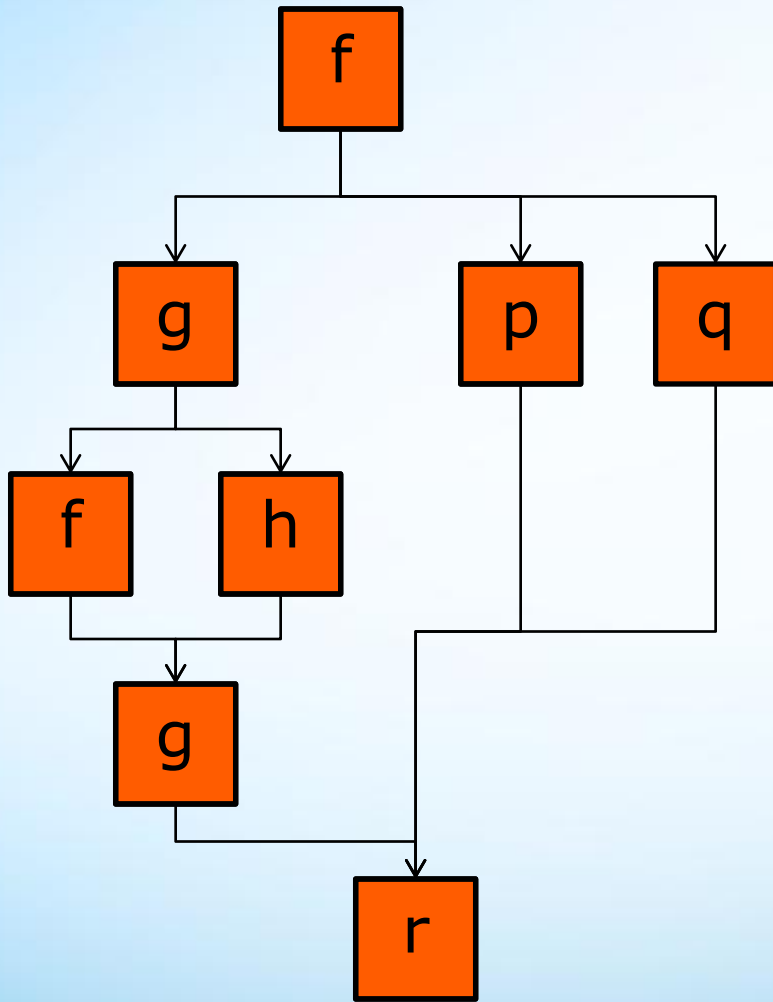
$$B = f(A) ;$$

$$C = g(B) ;$$

$$E = p(C) ;$$

$$F = q(A) ;$$

Superscalar Sequence (Task Graph)



Developer writes "serial" code:

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

```
C = g(B) ;
```

```
E = f(C) ;
```

```
F = h(C) ;
```

```
G = g(E, F) ;
```

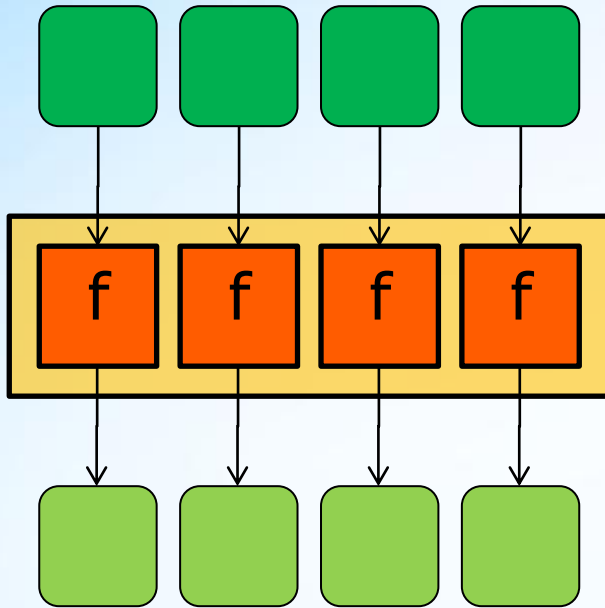
```
P = p(B) ;
```

```
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
- Variants: fork-join, general DAG

Map (Embarrassing Parallelism, SPMD)



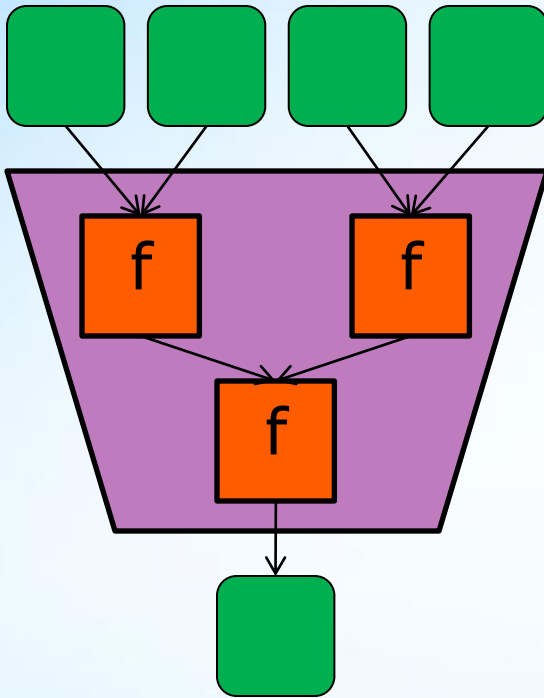
Examples: gamma correction and thresholding in images; color space conversions; Monte Carlo sampling; ray tracing.

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

$$A = \text{map}(f, B) ;$$

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

Reduction



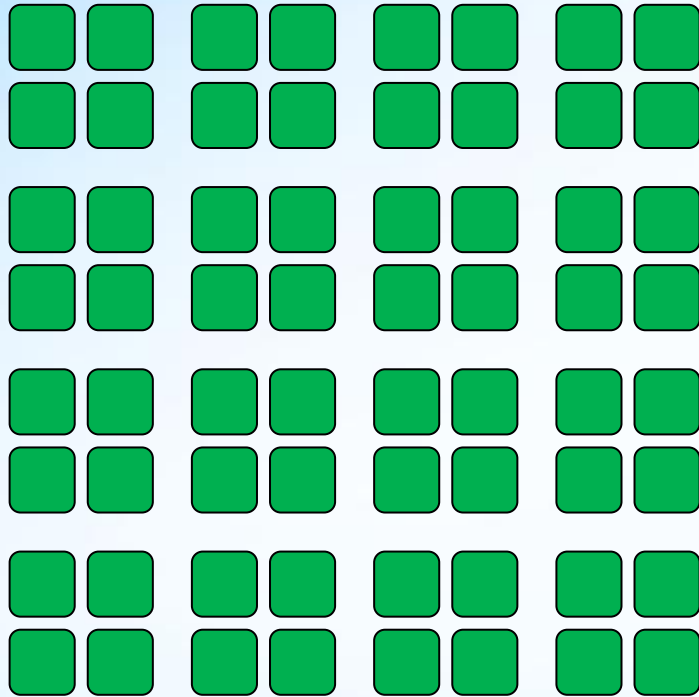
Examples: averaging of Monte Carlo samples; convergence testing; image comparison metrics; sub-task in matrix operations.

- *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*

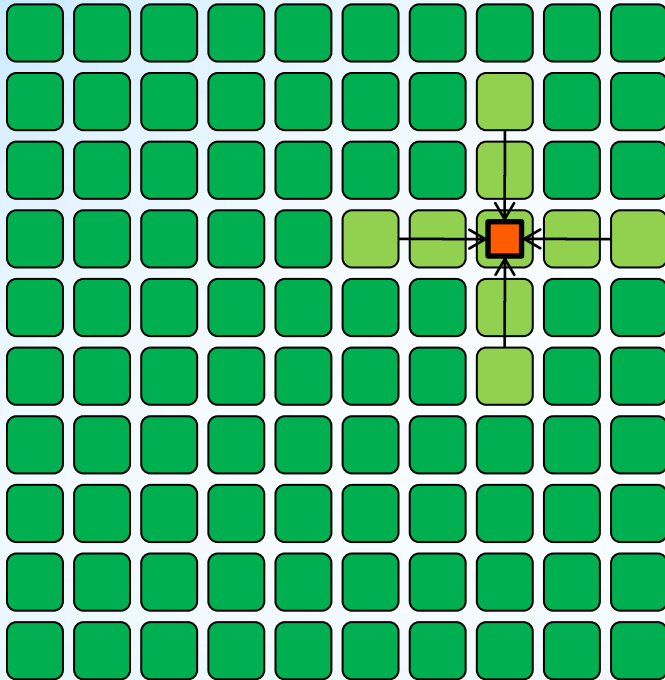
Partition (Geometric Decomposition)



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

- *Partition* breaks an input collection into a collection of collections
- Useful for divide-and-conquer algorithms
- Variants:
 - Uniform
 - Non-uniform
 - Overlapping (read-only)
- Issues:
 - How to deal with boundary conditions?
- Partitioning doesn't move data, it just provides an alternative "view" of its organization

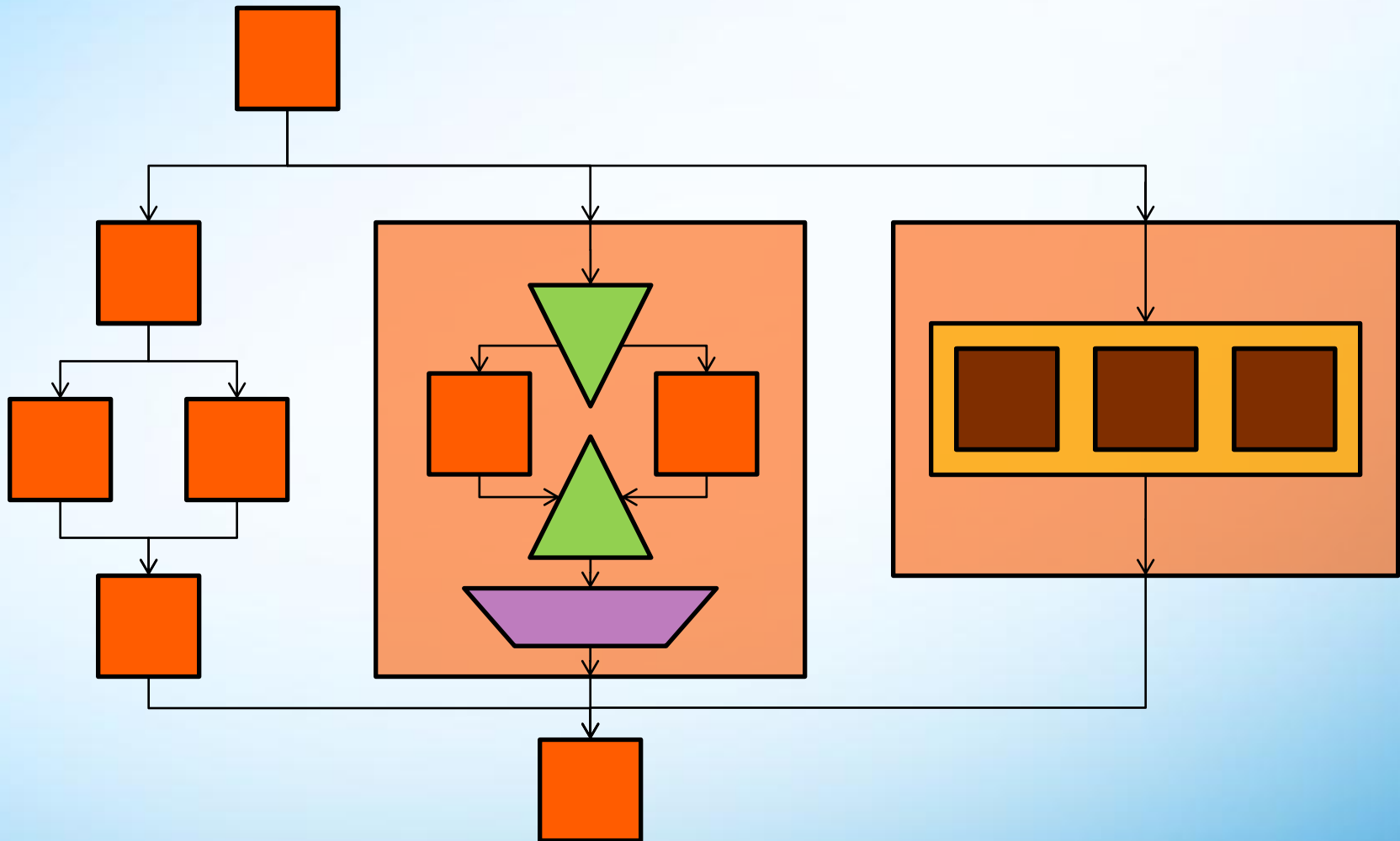
Stencil



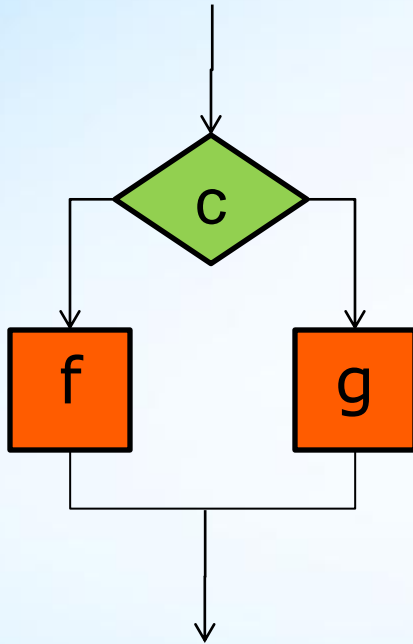
- *Stencil* applies function to all neighbourhoods of an array
- Neighbourhoods given by set of relative offsets
- Optimized implementation requires blocking and sliding windows
- Boundary conditions on array accesses need to be considered

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

Nesting: Recursive Composition



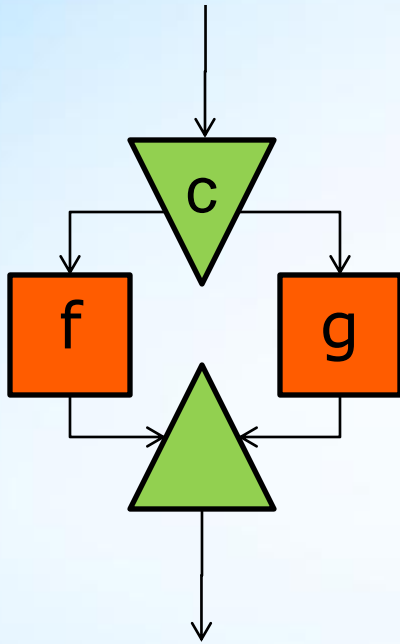
(Serial) Selection



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

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

Speculative Selection



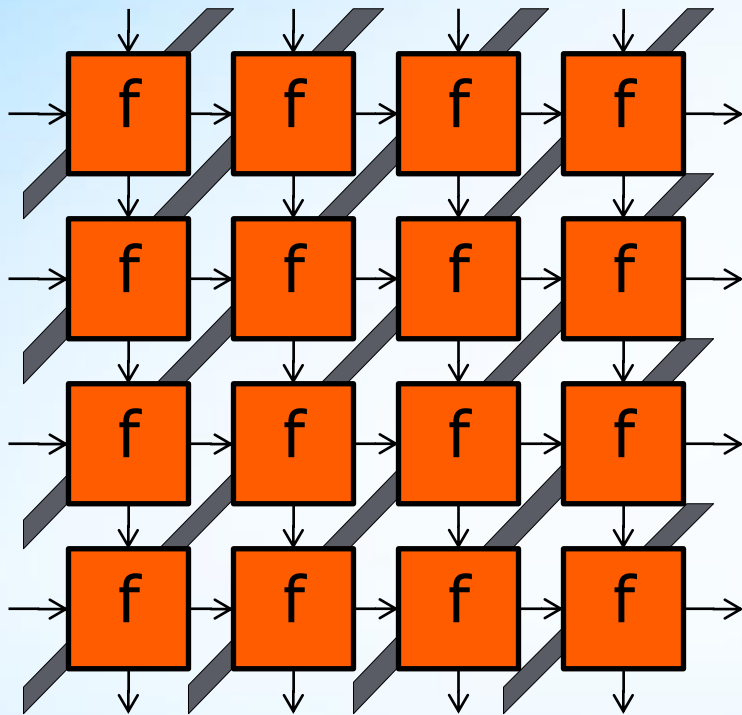
Examples: collision culling; ray tracing; clipping; discrete event simulation; search

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

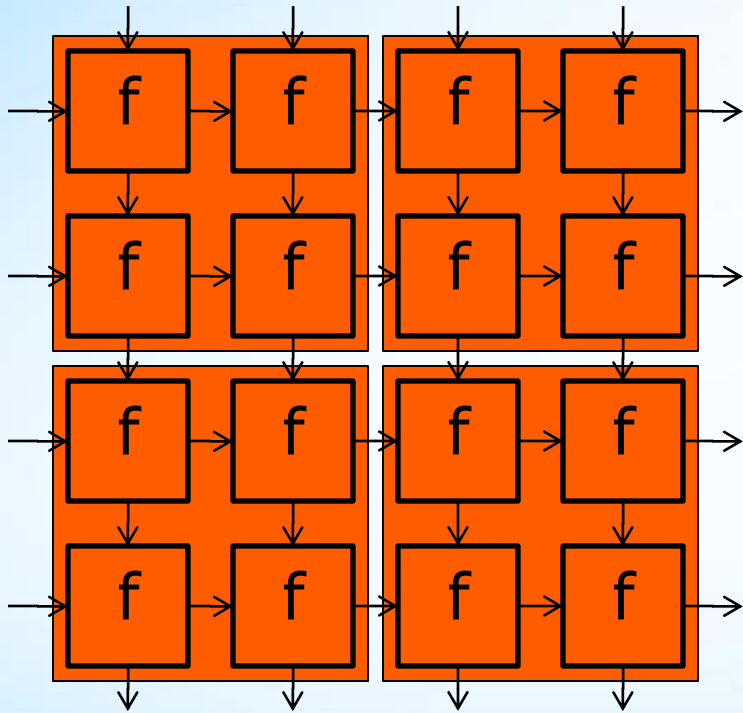
Recurrences



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

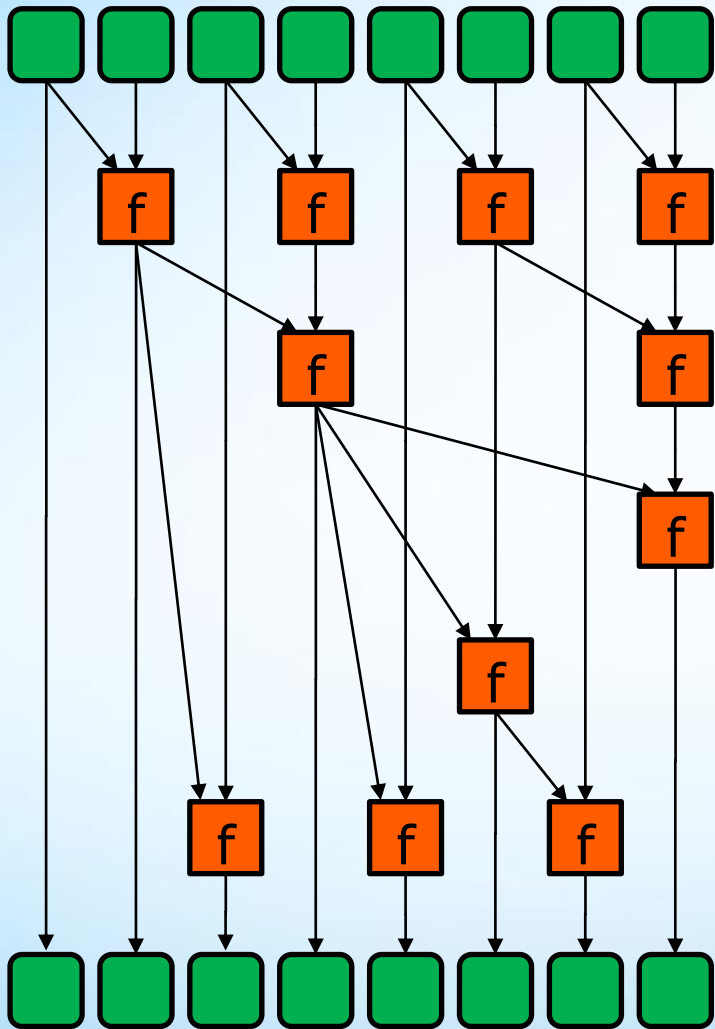
- Recurrences arise from the data dependency pattern given by nested loop-carried dependencies.
- nD recurrences can be parallelized over n-1 dimensions by Lamport's hyperplane theorem
- Execution of parallel slices can be performed either via iterative map, wavefront parallelism, or polyhedral decomposition

Partitioned (Blocked) Recurrences



- Implementation can use partitioning for higher performance
- When combined with the “pipeline” pattern recurrences implement “wavefront” computation.
- *Polyhedral theory generalizes this, can be used to drive recursive decompositions*

Scan

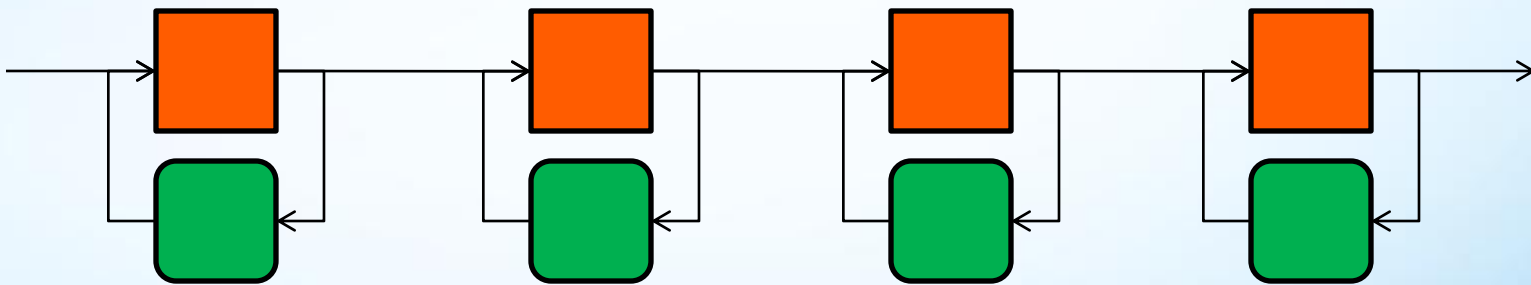


- Scan computes *all* partial reductions
- Allows parallelization of many 1D recurrences
- Requires an associative operator
- Requires $2n$ work over serial execution, but $\lg n$ steps

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

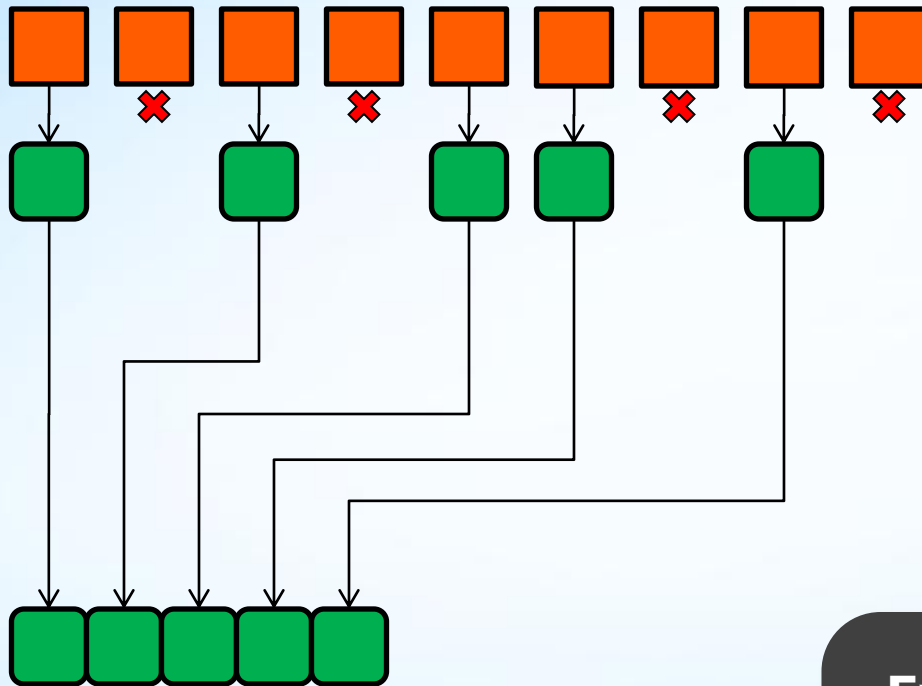
Pipeline

- 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.

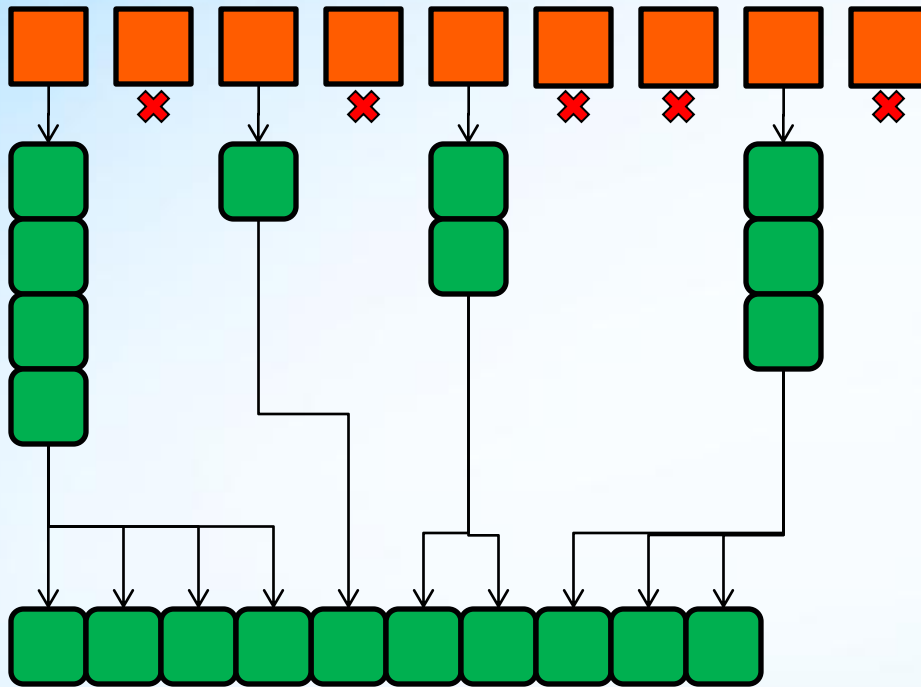
Pack



- 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.

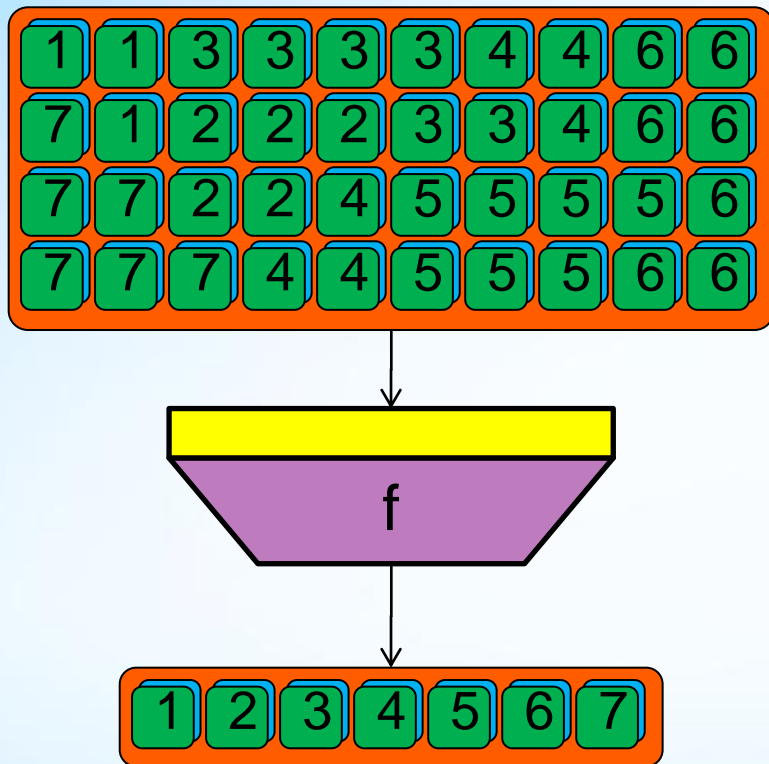
Expand



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

- 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

Search/Match



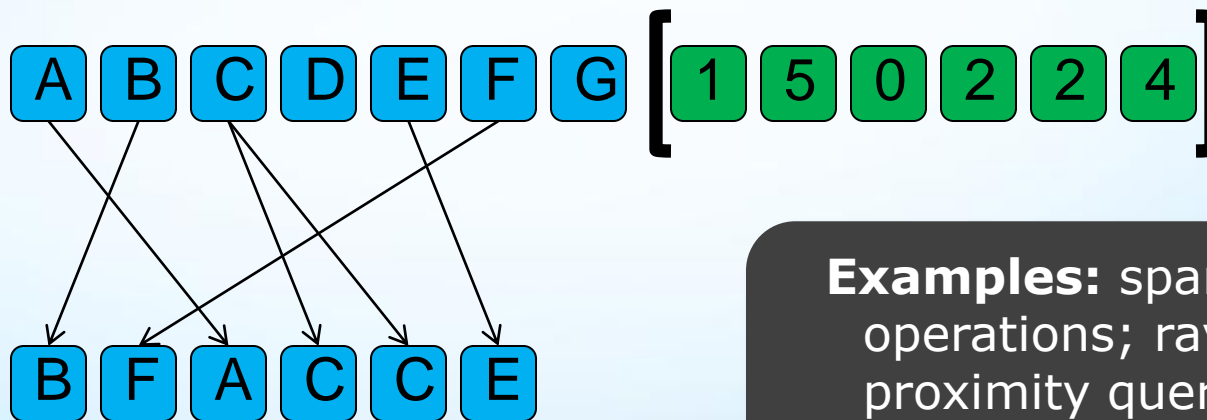
Examples: computation of metrics on segmented regions in vision; computation of web analytics

- Searching and matching fundamental capabilities; may depend indirectly on sorting or hashing
- 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

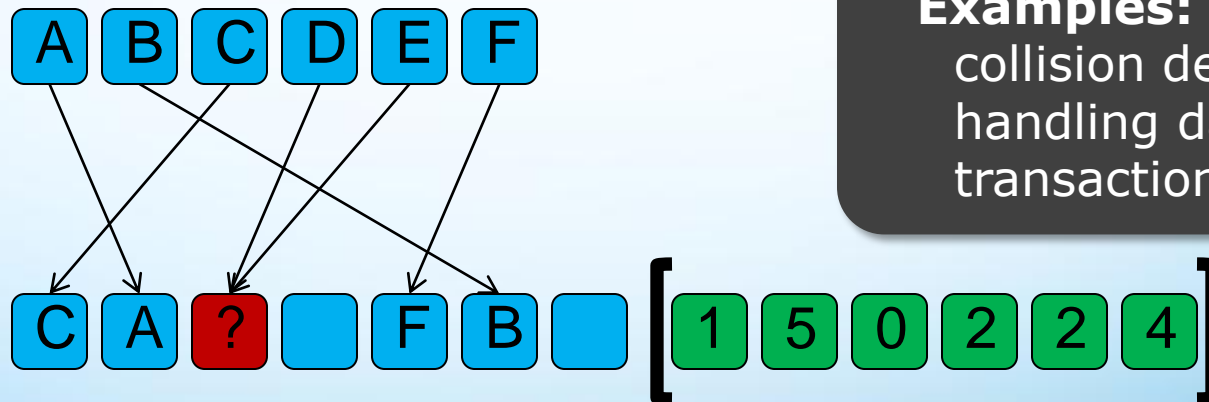


Examples: sparse matrix operations; ray tracing; proximity queries; collision detection.

!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

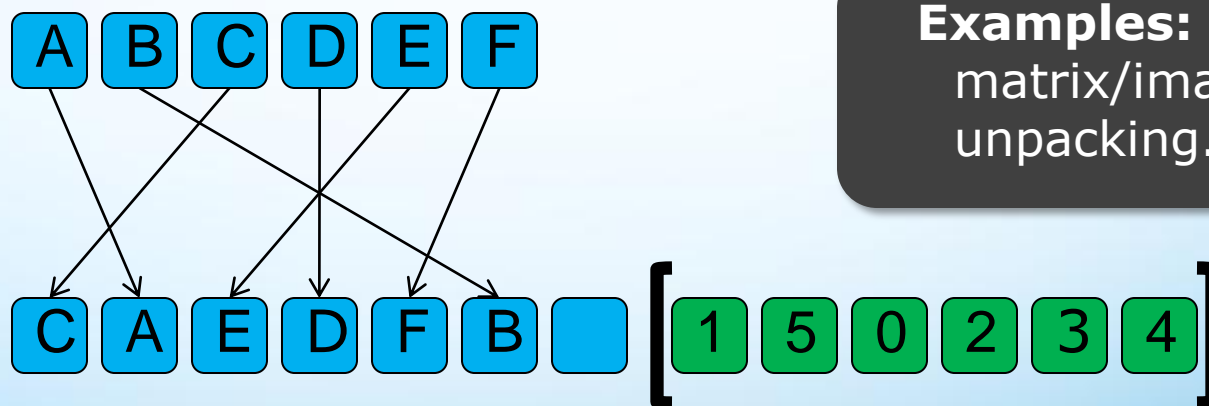


Examples: marking pairs in collision detection; handling database update transactions.

*Permutation Scatter

Option 1: 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"

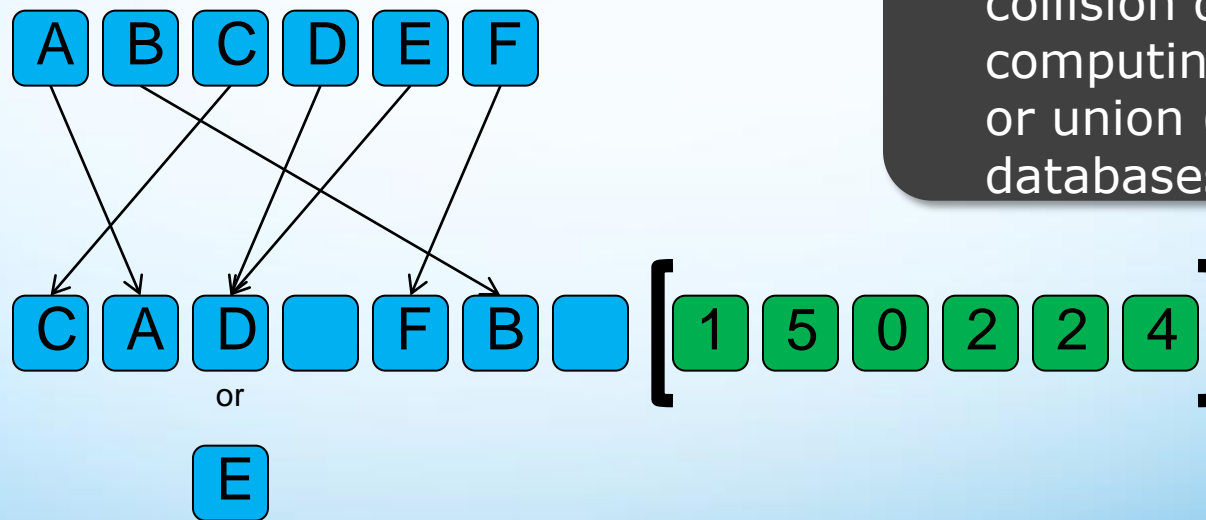


Examples: FFT scrambling;
matrix/image transpose;
unpacking.

!Atomic Scatter

Option 2: 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

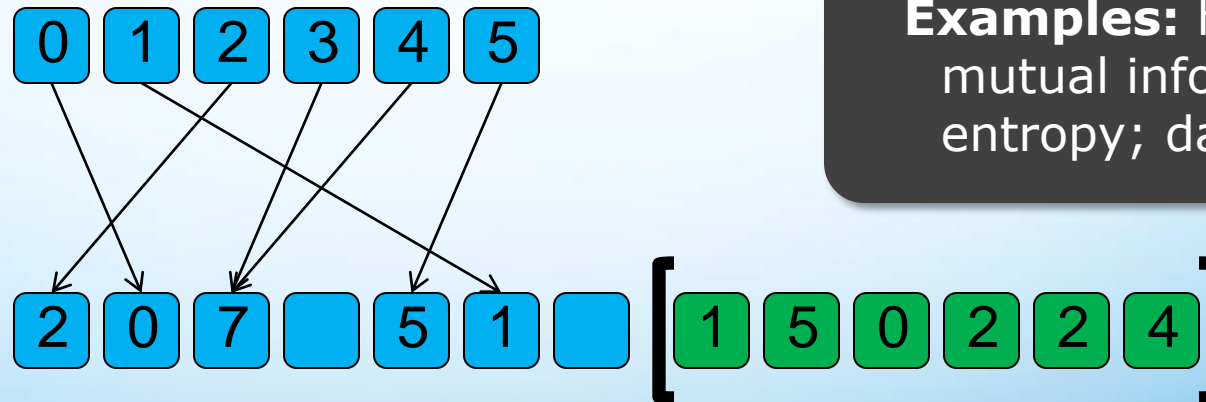


Examples: marking pairs in collision detection; computing set intersection or union (used in text databases)

Merge Scatter

Option 3: 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.



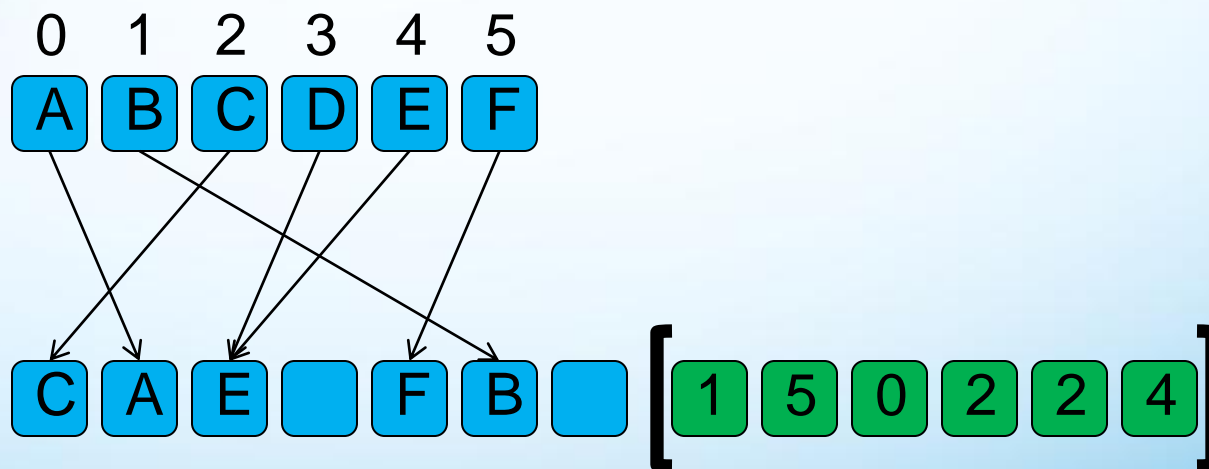
Examples: histogram;
mutual information and
entropy; database updates.

Priority Scatter

Option 4: 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...



Other Parallel Patterns

Fork-join: special case of nesting

Workpile: extension of map where tasks can be added dynamically

Branch-and-bound: *non-deterministic* search where other branches can be terminated once once a “good enough” solution found

Incremental graph update: propagation of updates through DAG or graph (latter may not terminate, however...)

Graph rewriting: can be used to implement functional languages.

Transactions: non-deterministic database updates