Structured Parallel Programming with Patterns

SC13 Tutorial
Sunday, November 17th
8:30am to 5pm

Michael Hebenstreit
James R. Reinders
Arch Robison
Michael McCool

Course Outline

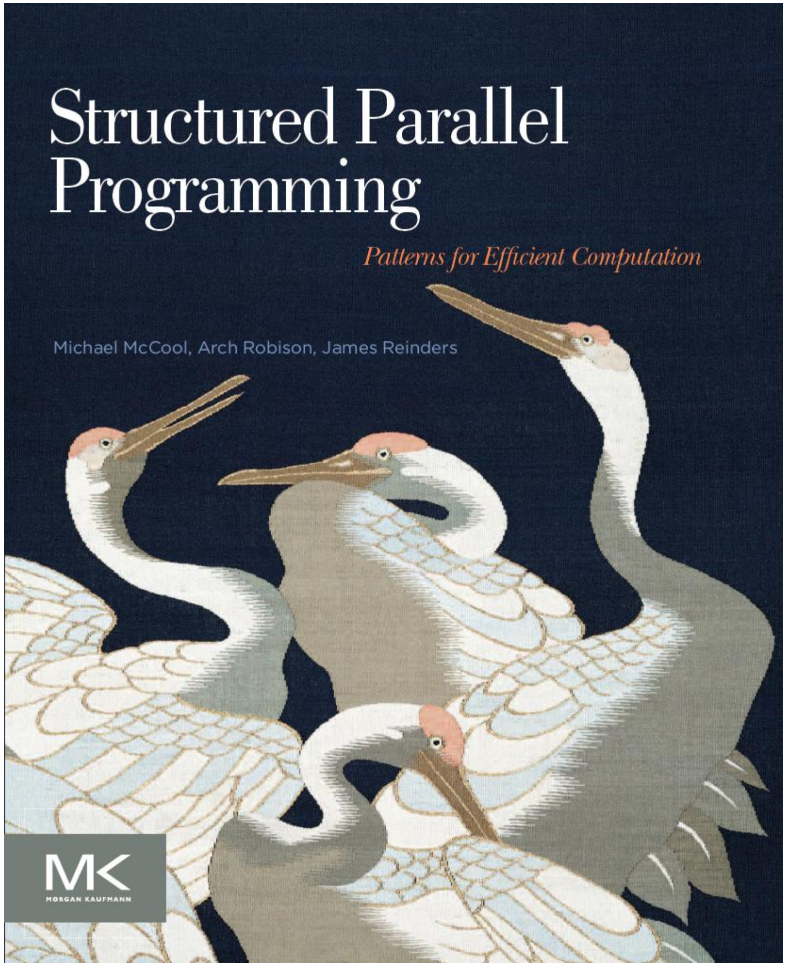
- Introduction
  - Motivation and goals
  - Patterns of serial and parallel computation
- Background
  - Machine model
  - Complexity, work-span
- Intel® Cilk™ Plus and Intel® Threading Building Blocks (Intel® TBB)
  - Programming models
  - Examples

Text

Structured Parallel Programming: Patterns for Efficient Computation

- Michael McCool
- Arch Robison
- James Reinders

- Uses Cilk Plus and TBB as primary frameworks for examples.
- Appendices concisely summarize Cilk Plus and TBB.
- www.parallelbook.com



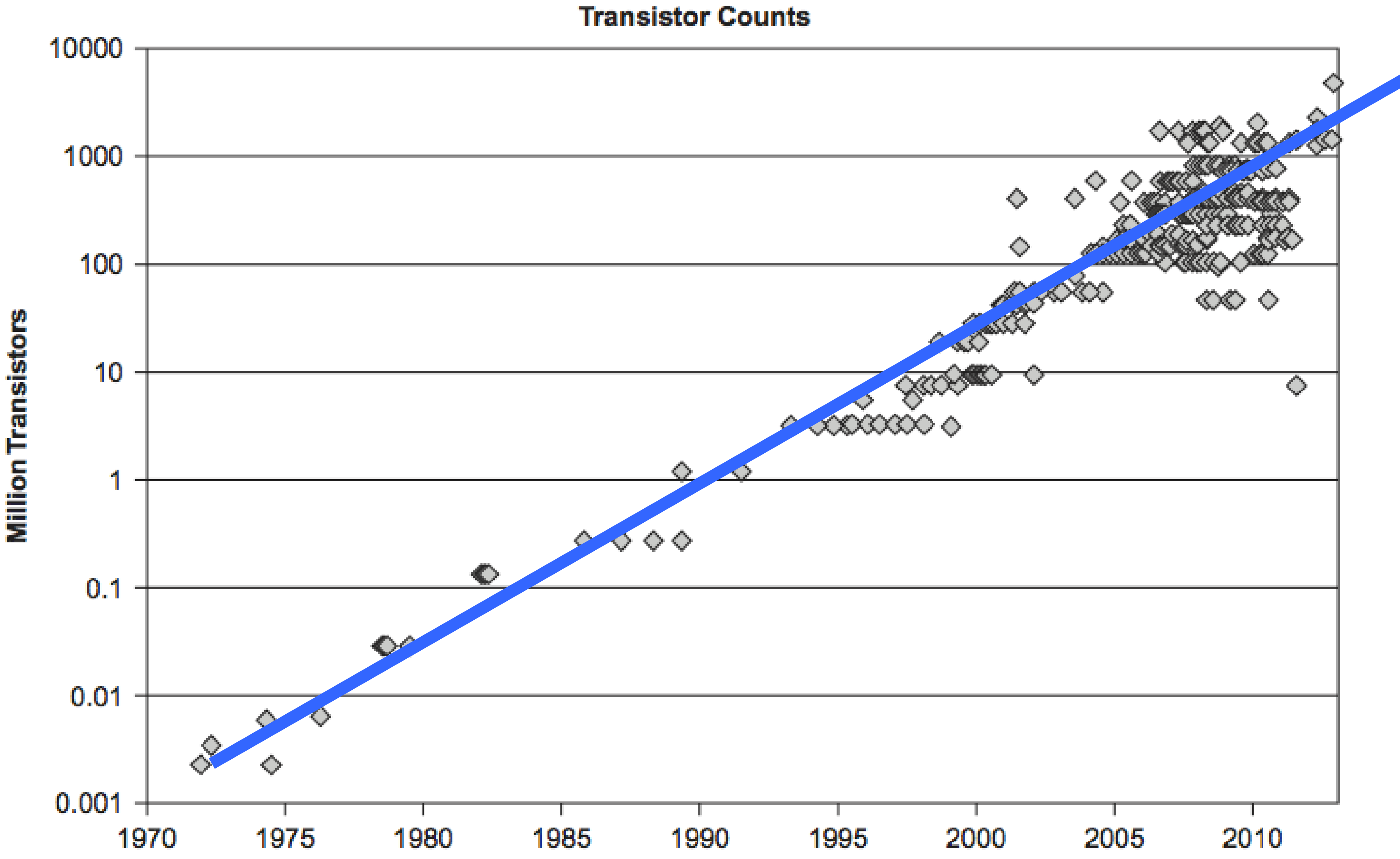
INTRODUCTION

Introduction: Outline

- Evolution of Hardware to Parallelism
- Software Engineering Considerations
- Structured Programming with Patterns
- Parallel Programming Models
- Simple Example: Dot Product

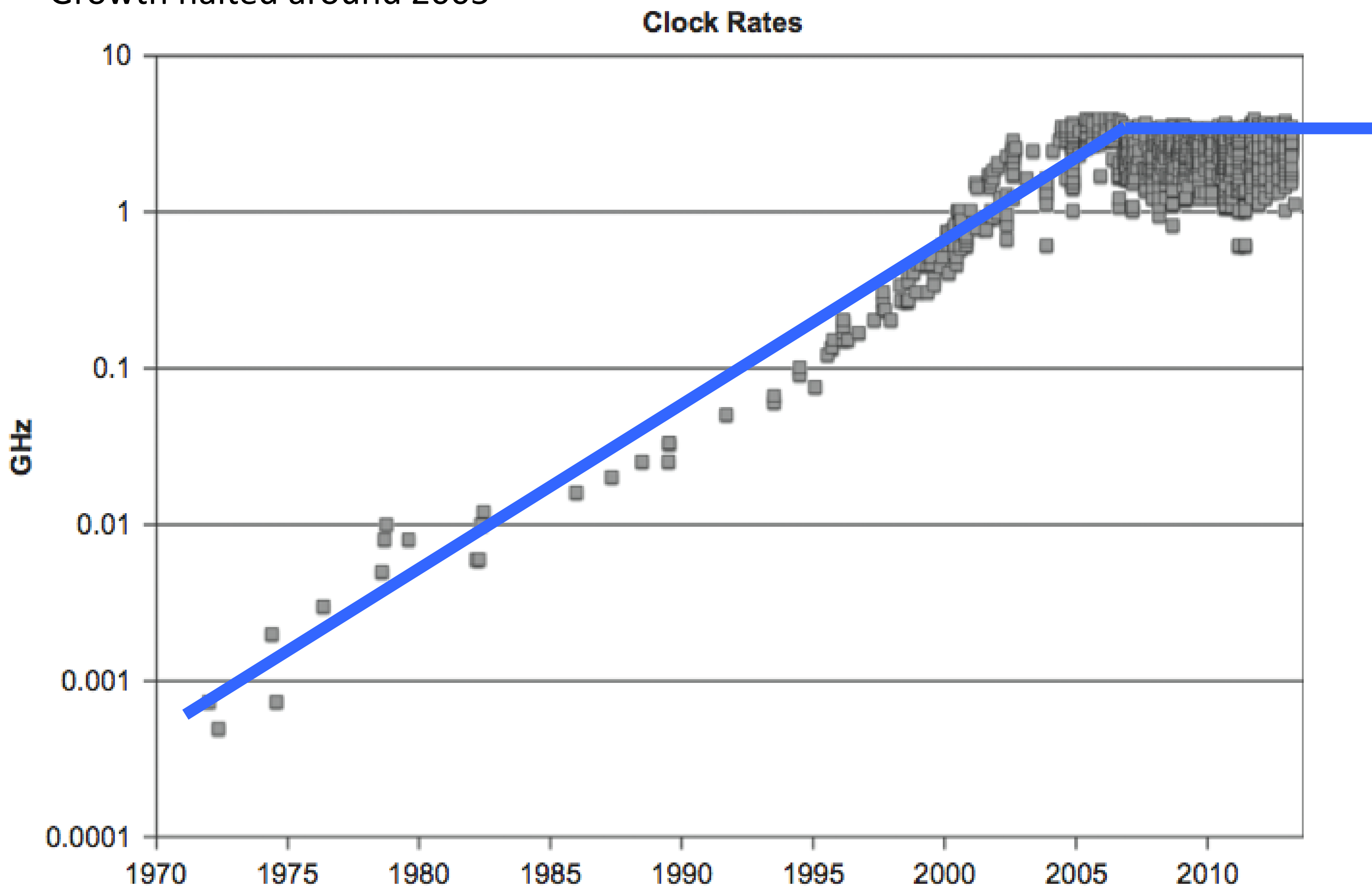
Transistors per Processor over Time

Continues to grow exponentially (Moore's Law)



Processor Clock Rate over Time

Growth halted around 2005



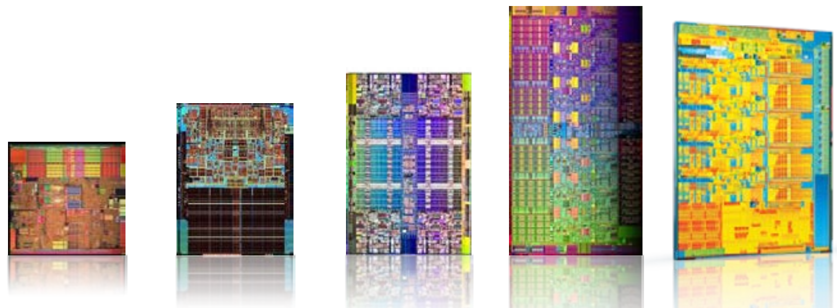
Hardware Evolution

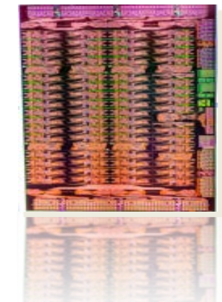
There are limits to "automatic" improvement of scalar performance:

1. **The Power Wall:** Clock frequency cannot be increased without exceeding air cooling.
2. **The Memory Wall:** Access to data is a limiting factor.
3. **The ILP Wall:** All the existing instruction-level parallelism (ILP) is already being used.

→ **Conclusion:** Explicit parallel mechanisms and explicit parallel programming are *required* for performance scaling.

Trends: More cores. Wider vectors. Coprocessors.

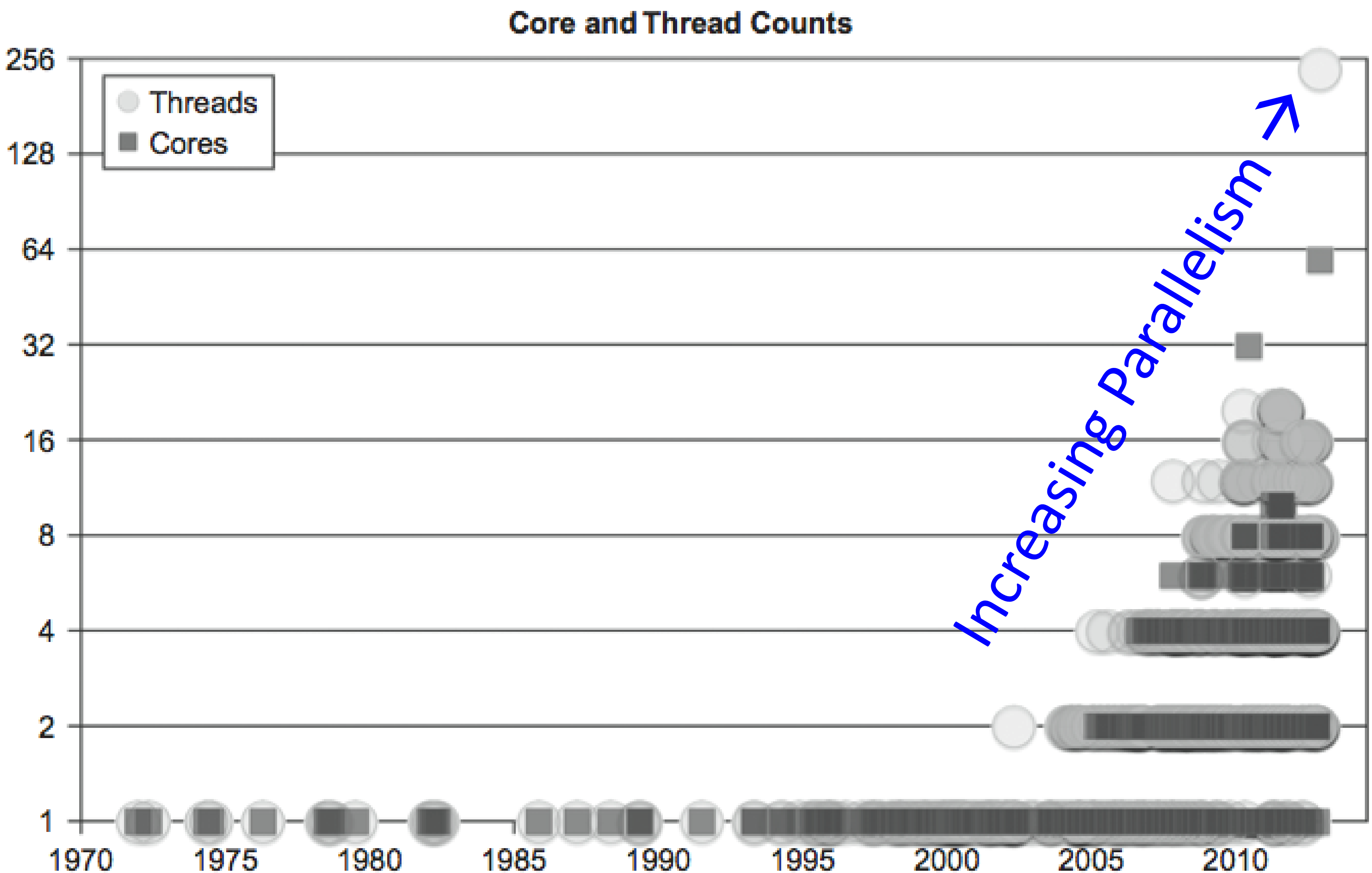


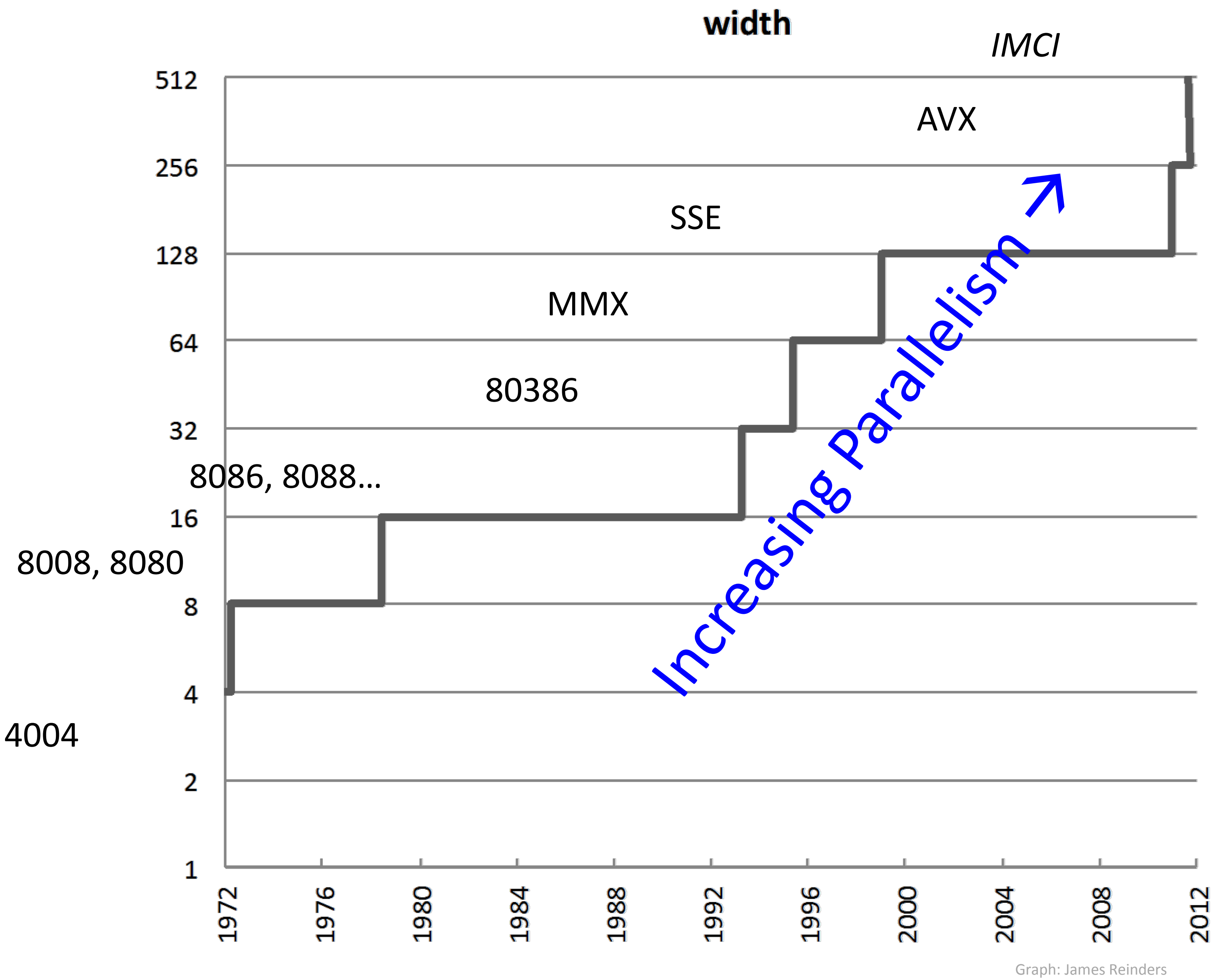


Images do not reflect actual die sizes

	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge	Intel® Xeon® processor code-named Ivy Bridge	Intel® Xeon® processor code-named Haswell	Intel® Xeon Phi™ coprocessor code-named Knights Corner
Core(s)	1	2	4	6	8			57-61
Threads	2	2	8	12	16			228-244
SIMD Width	128	128	128	128	256	256	256	512
	SSE2	**SSSE3**	**SSE4.2**	**SSE4.2**	**AVX**	**AVX**	**AVX2 FMA3**	**IMCI**

Software challenge: Develop scalable software


Core and Thread Counts
Threads
Cores
256
128
64
32
16
8
4
2
1
1970
1975
1980
1985
1990
1995
2000
2005
2010
Increasing Parallelism


width
IMCI
AVX
SSE
MMX
80386
8086, 8088…
8008, 8080
4004
Increasing Parallelism
512
256
128
64
32
16
8
4
2
1
1972
1976
1980
1984
1988
1992
1996
2000
2004
2008
2012
Graph: James Reinders

Parallelism and Performance

There are limits to "automatic" improvement of scalar performance:

1. **The Power Wall:** Clock frequency cannot be increased without exceeding air cooling.
2. **The Memory Wall:** Access to data is a limiting factor.
3. **The ILP Wall:** All the existing instruction-level parallelism (ILP) is already being used.

→ **Conclusion:** Explicit parallel mechanisms and explicit parallel programming are *required* for performance scaling.

Parallel SW Engineering Considerations

- **Problem:** Amdahl's Law* notes that scaling will be limited by the serial fraction of your program.
- ***Solution:*** *scale the parallel part of your program faster than the serial part using data parallelism.*
- **Problem:** Locking, access to data (memory and communication), and overhead will strangle scaling.
- ***Solution:*** *use programming approaches with good data locality and low overhead, and avoid locks.*
- **Problem:** Parallelism introduces new debugging challenges: deadlocks and race conditions.
- ***Solution:*** *use structured programming strategies to avoid these by design, improving maintainability.*

*Except Amdahl was an optimist, as we will discuss.

PATTERNS

Structured Programming with Patterns

- Patterns are “best practices” for solving specific problems.
- Patterns can be used to organize your code, leading to algorithms that are more scalable and maintainable.
- A pattern supports a particular “algorithmic structure” with an efficient implementation.
- Good parallel programming models support a set of useful parallel patterns with low-overhead implementations.

Structured Serial Patterns

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

- Sequence
- Selection
- Iteration
- Nesting
- Functions
- Recursion
- Random read
- Random write
- Stack allocation
- Heap allocation
- Objects
- Closures

Using these patterns, “goto” can (mostly) be eliminated and the maintainability of software improved.

Structured Parallel Patterns

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

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

Using these patterns, threads and vector intrinsics can (mostly) be eliminated and the maintainability of software improved.

Some Basic Patterns

- **Serial:** Sequence

→ **Parallel:** Superscalar Sequence

- **Serial:** Iteration

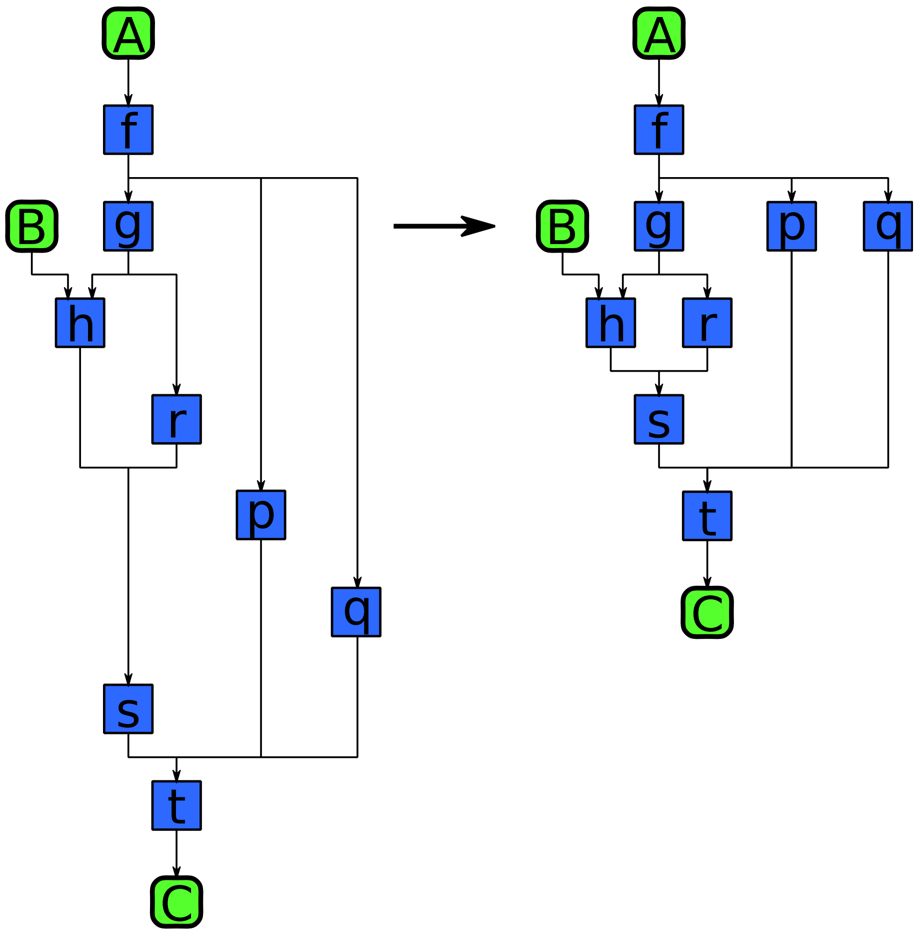
→ **Parallel:** Map, Reduction, Scan, Recurrence…

(Serial) Sequence

A serial sequence is executed in the exact order given:

```
F = f(A);
G = g(F);
B = h(G);
```

Superscalar Sequence

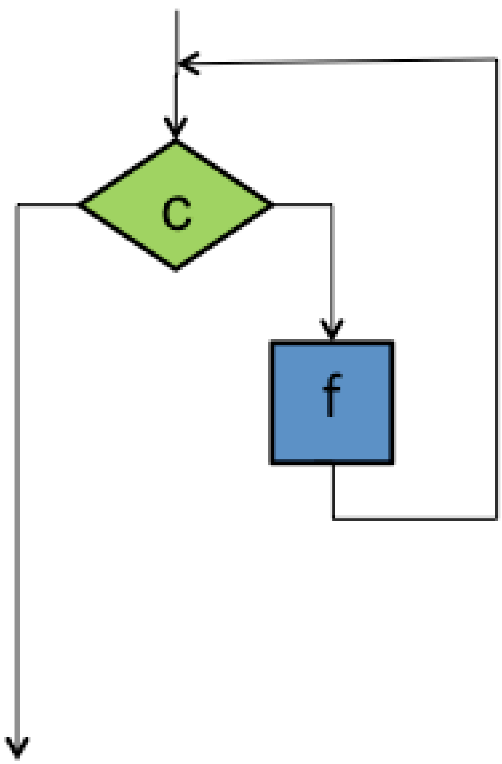


Developer writes "serial" code:

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

- Tasks ordered only by data dependencies
- Tasks can run whenever input data is ready

(Serial) Iteration



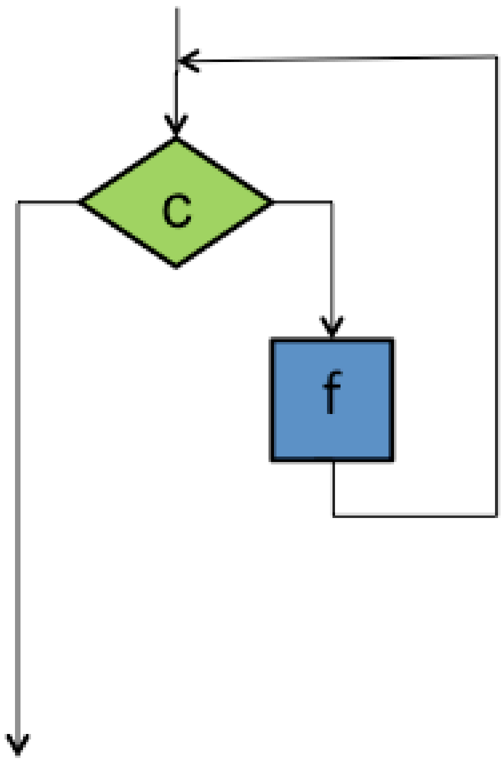
The iteration pattern repeats some section of code as long as a condition holds

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

Each iteration can depend on values computed in any earlier iteration.

The loop can be terminated at any point based on computations in any iteration

(Serial) Countable Iteration



The iteration pattern repeats some section of code a specific number of times

```
for (i = 0; i<n; ++i) {
    f();
}
```

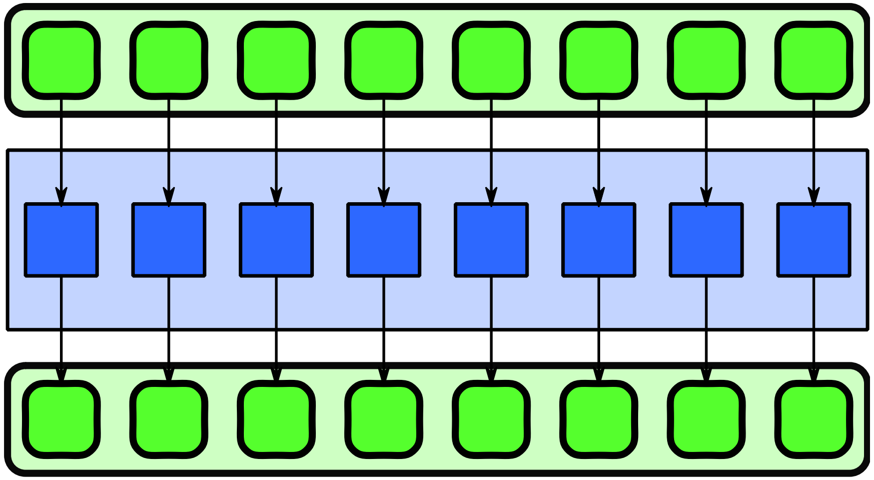
This is the same as

```
i = 0;
while (i<n) {
    f();
    ++i;
}
```

Parallel "Iteration"

- The serial iteration pattern actually maps to several *different* parallel patterns
- It depends on whether and how iterations depend on each other…
- Most parallel patterns arising from iteration require a fixed number of invocations of the body, known in advance

Map



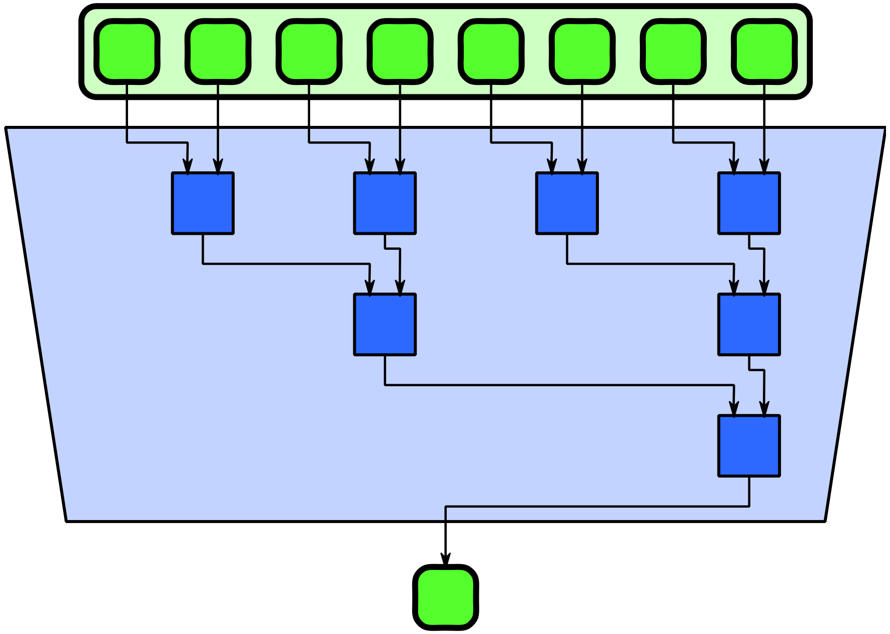
Examples: gamma correction and thresholding in images; color space conversions; Monte Carlo sampling; ray tracing.

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

```
for (i=0; i<n; ++i) {
    f(A[i]);
}
```

- Map replaces *one specific* usage of iteration in serial programs: *independent operations.*

Reduction



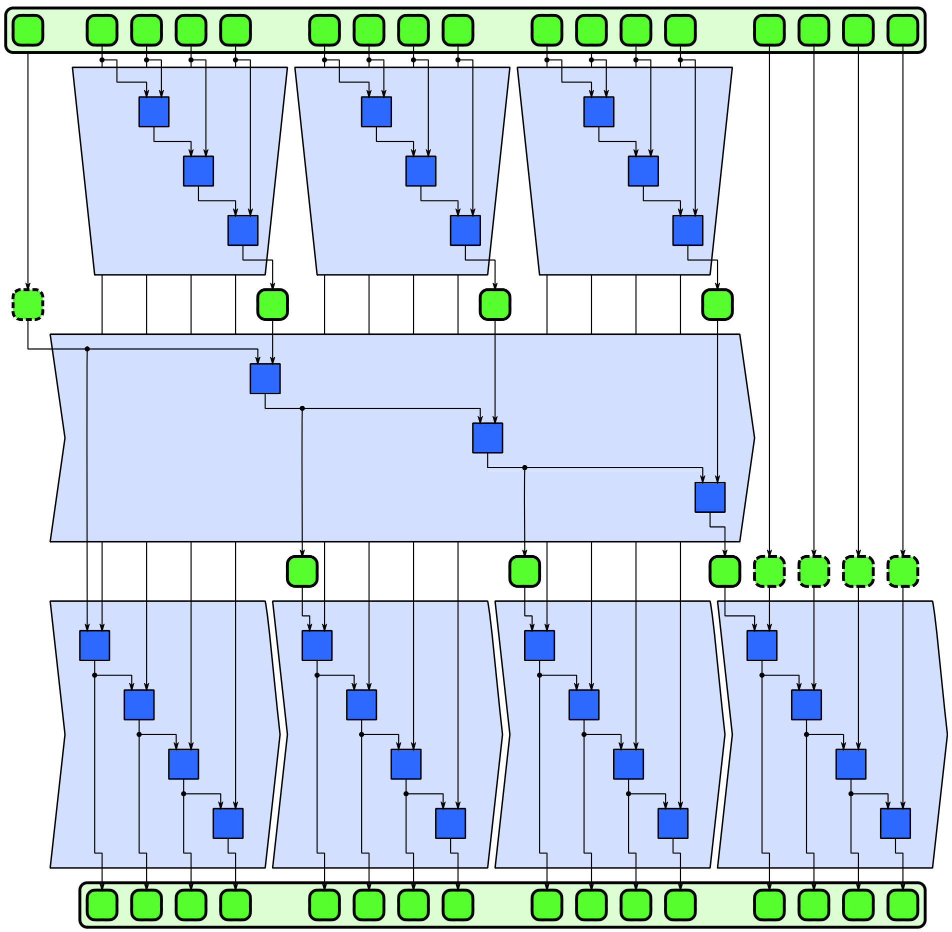
- *Reduction* combines every element in a collection into one element using an associative operator.

```
b = 0;
for (i=0; i<n; ++i) {
    b += f(B[i]);
}
```

- Reordering of the operations is often needed to allow for parallelism.
- A tree reordering requires associativity.

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

Scan



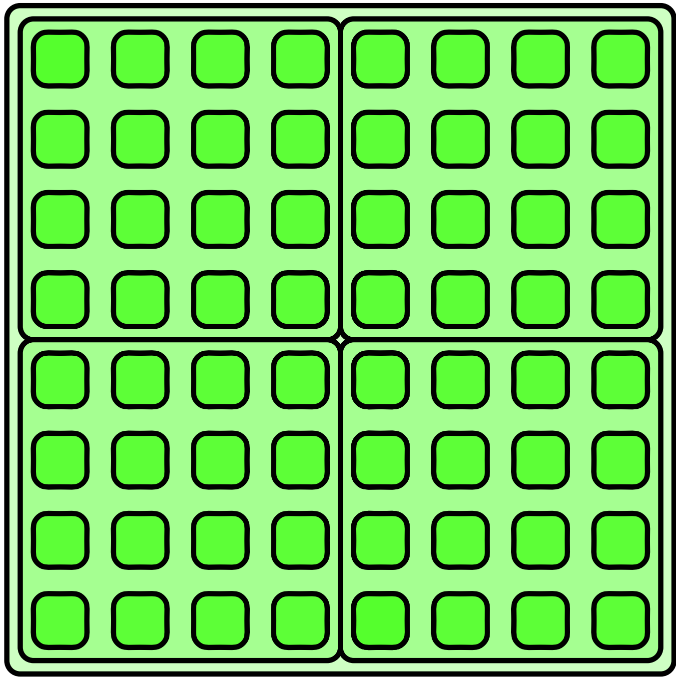
- *Scan* computes all partial reductions of a collection

```
A[0] = B[0] + init;
for (i=1; i<n; ++i) {
 A[i] = B[i] + A[i-1];
}
```

- Operator must be (at least) associative.
- Diagram shows one possible parallel implementation using three-phase strategy

Examples: random number generation, pack, tabulated integration, time series analysis

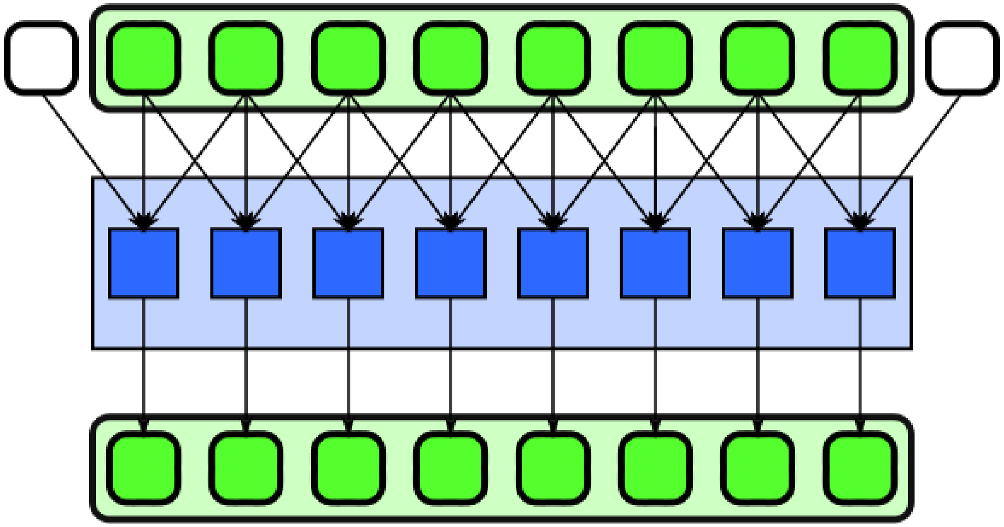
Geometric Decomposition/Partition



- *Geometric decomposition* breaks an input collection into sub-collections
- *Partition* is a special case where sub-collections do not overlap
- Does not move data, it just provides an alternative "view" of its organization

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

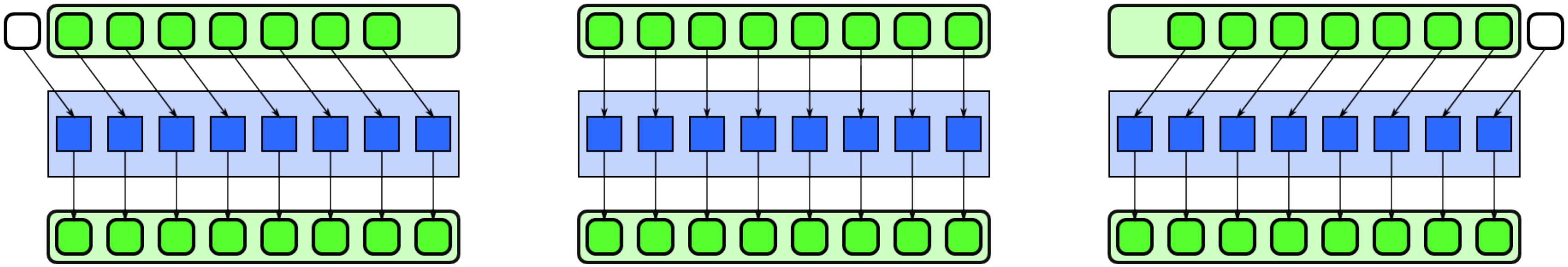
Stencil

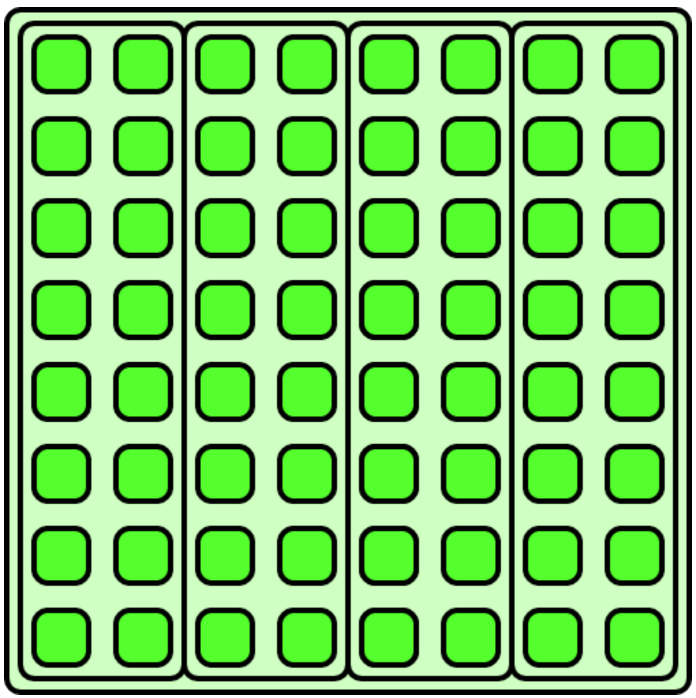


Examples: signal filtering including convolution, median, anisotropic diffusion

- *Stencil* applies a function to neighbourhoods of a collection.
- Neighbourhoods are given by set of relative offsets.
- Boundary conditions need to be considered, but majority of computation is in interior.

Implementing Stencil

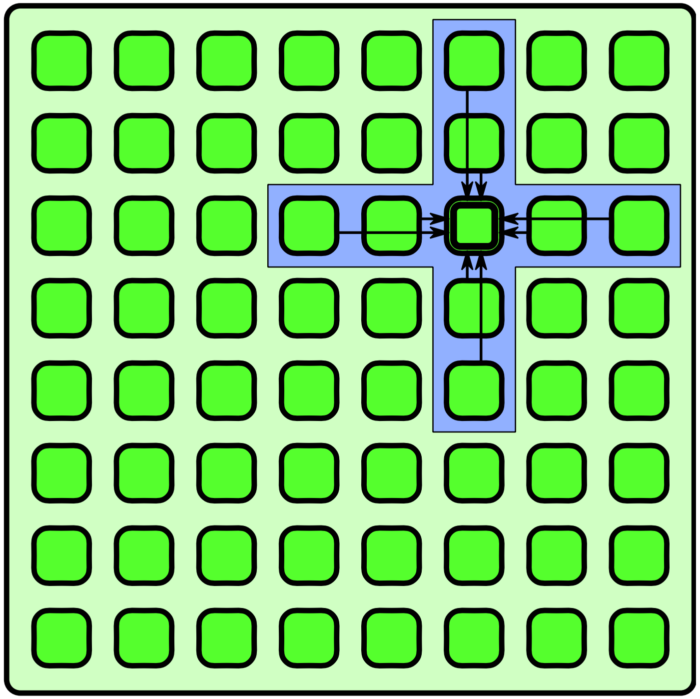




Vectorization can include converting regular reads into a set of shifts.

Strip-mining reuses previously read inputs within serialized chunks.

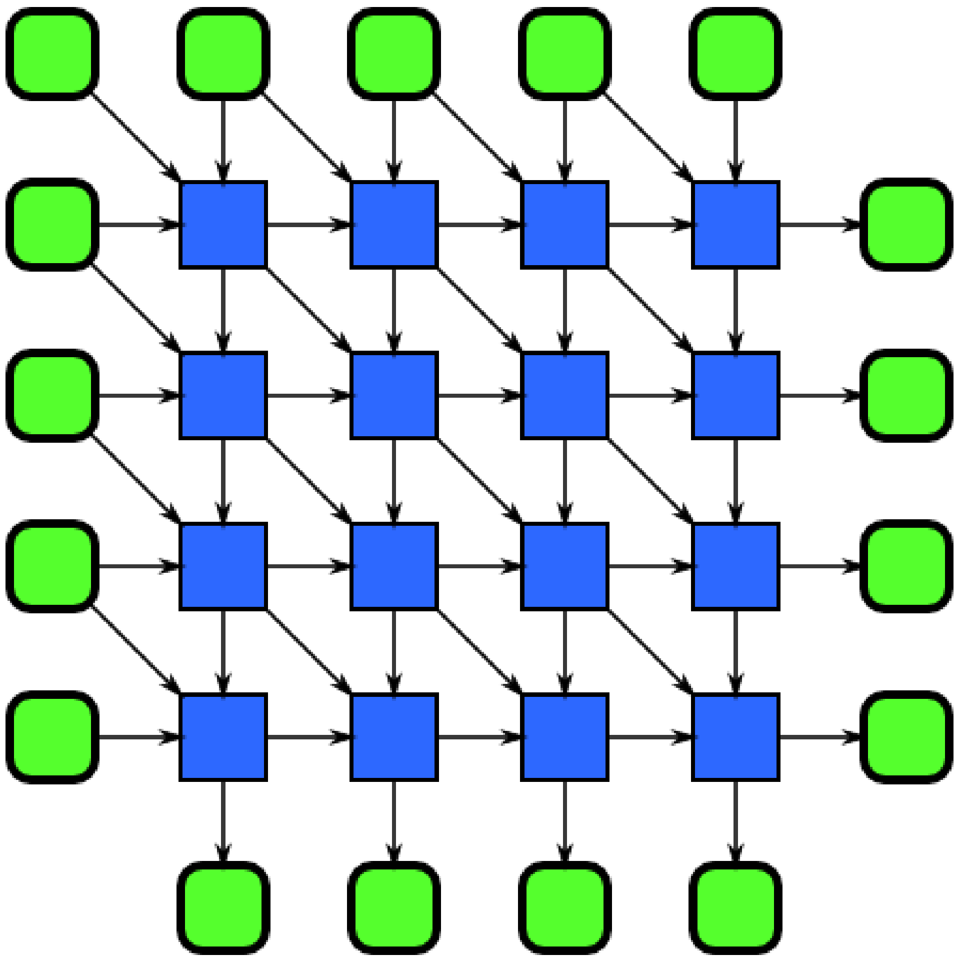
nD Stencil



- *nD Stencil* applies a function to neighbourhoods of an nD array
- Neighbourhoods are given by set of relative offsets
- Boundary conditions need to be considered

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

Recurrence



- *Recurrence* results from loop nests with both input and output dependencies between iterations

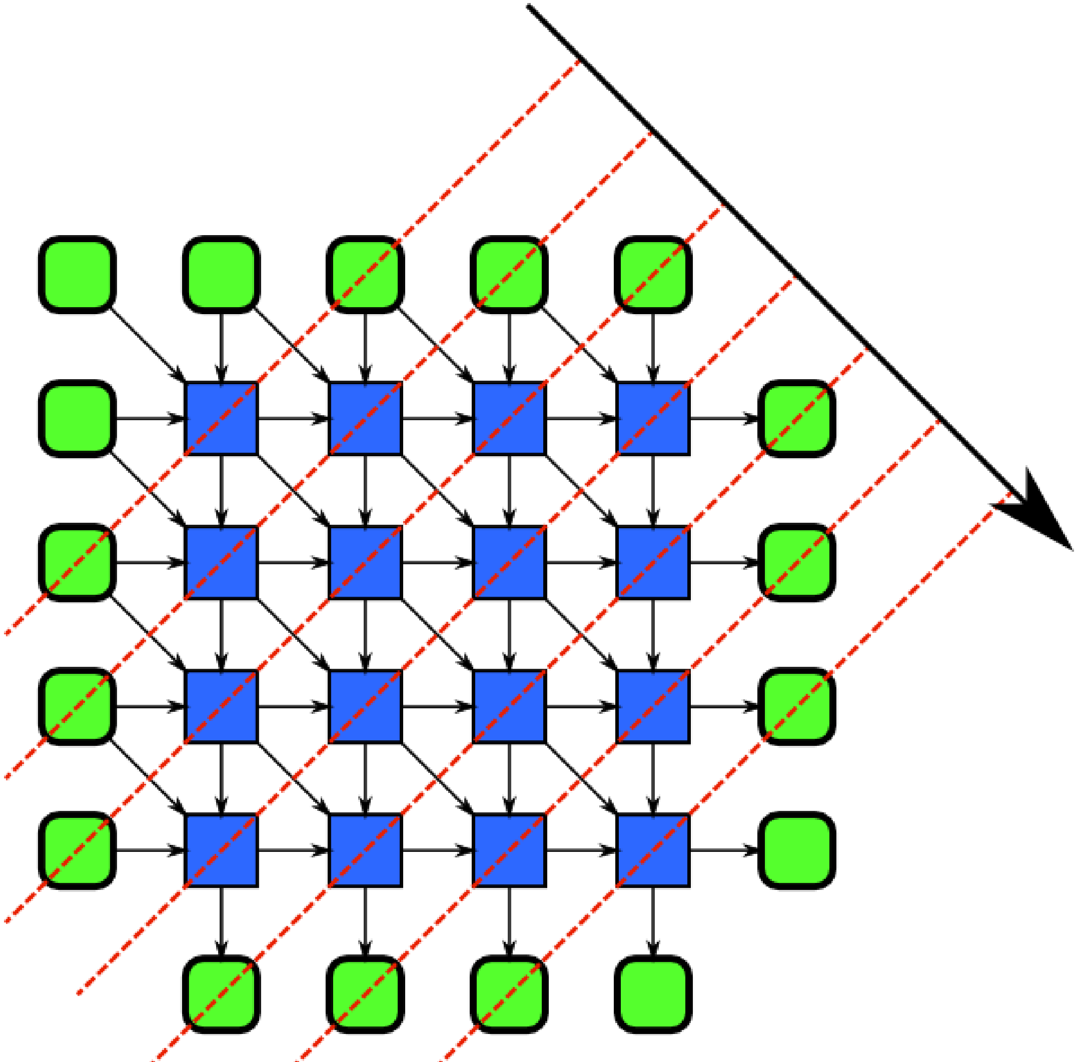
- Can also result from iterated stencils

Examples: Simulation including fluid flow, electromagnetic, and financial PDE solvers, lattice QCD, sequence alignment and pattern matching

Recurrence

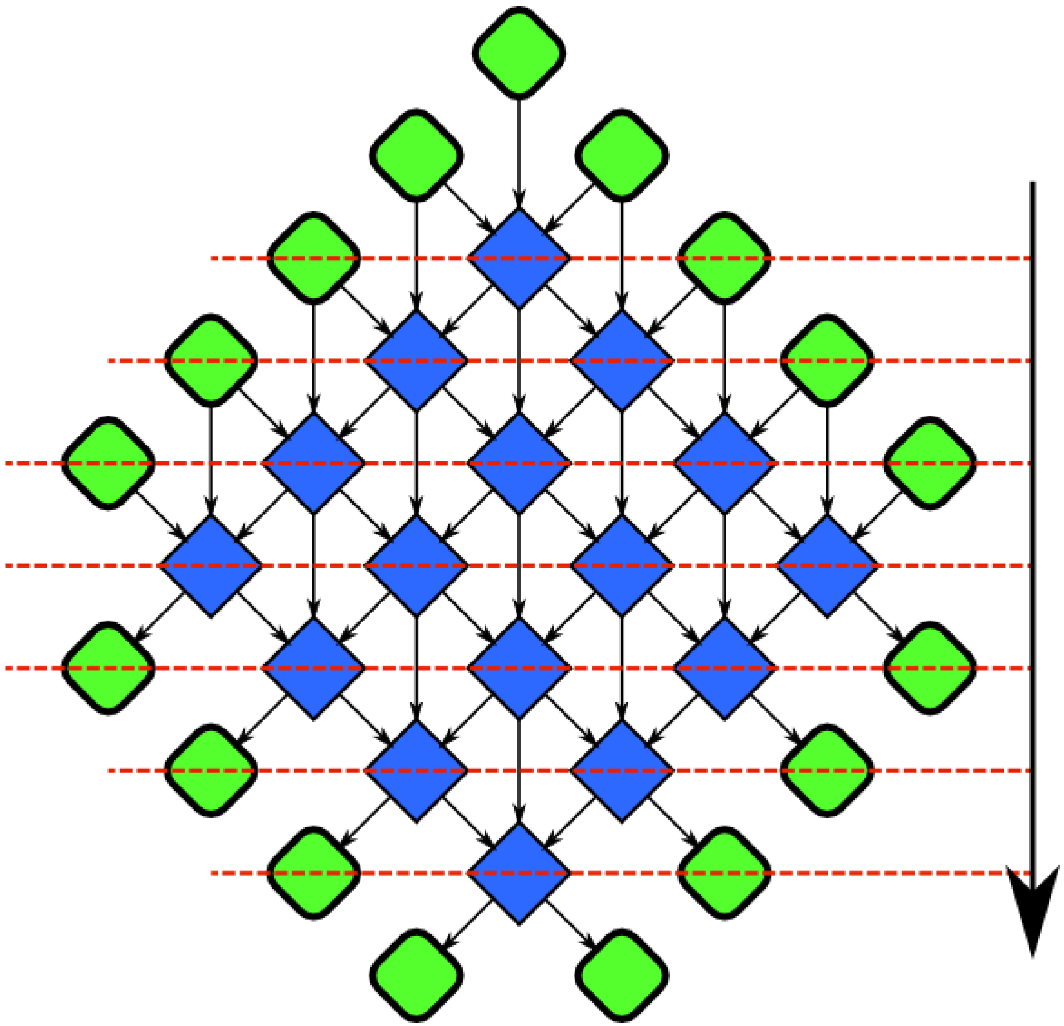
```
for (int i = 1; i < N; i++) {
   for (int j = 1; j < M; j++) {
      A[i][j] = f(
         A[i-1][j],
         A[i][j-1],
         A[i-1][j-1],
         B[i][j]);
    }
}
```

Recurrence Hyperplane Sweep



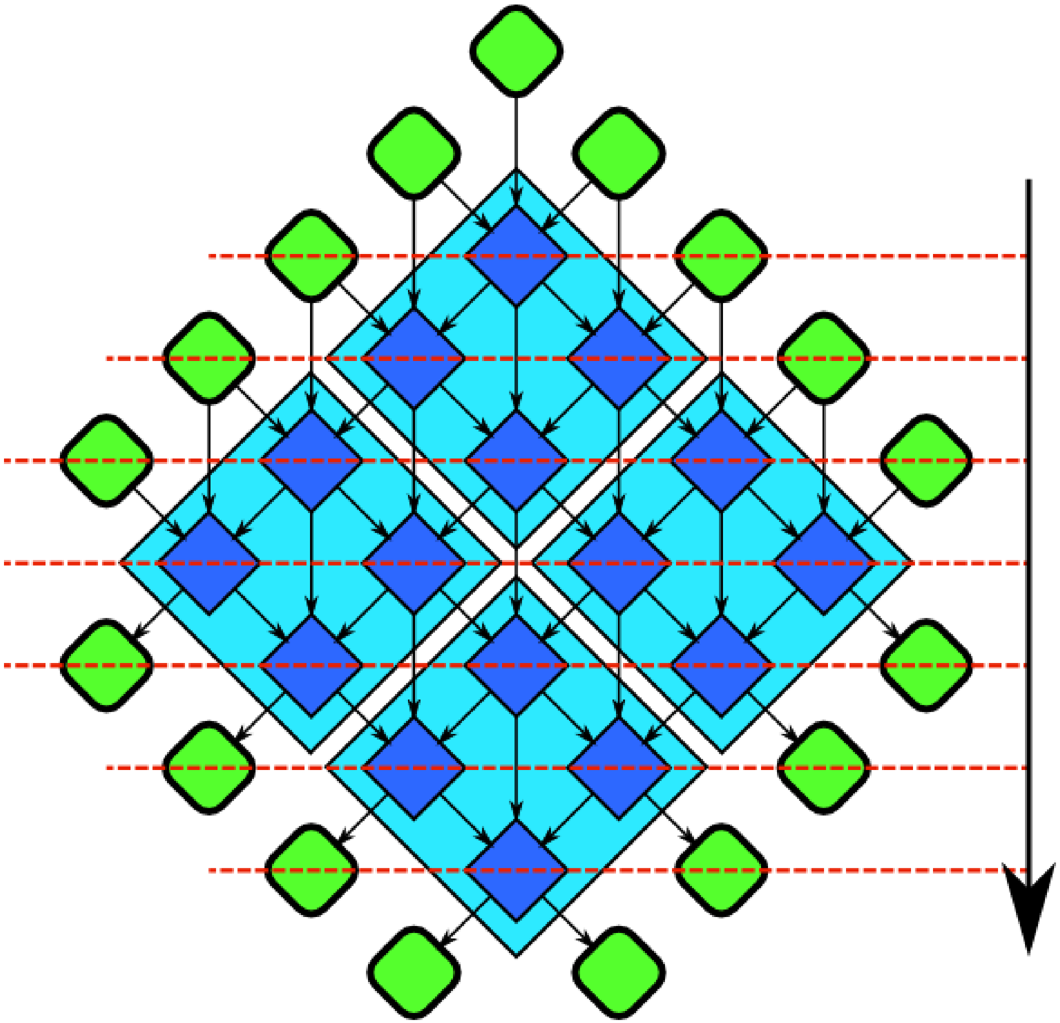
- Multidimensional recurrences can *always* be parallelized
- Leslie Lamport's hyperplane separation theorem:
  - Choose hyperplane with inputs and outputs on opposite sides
  - Sweep through data perpendicular to hyperplane

Rotated Recurrence



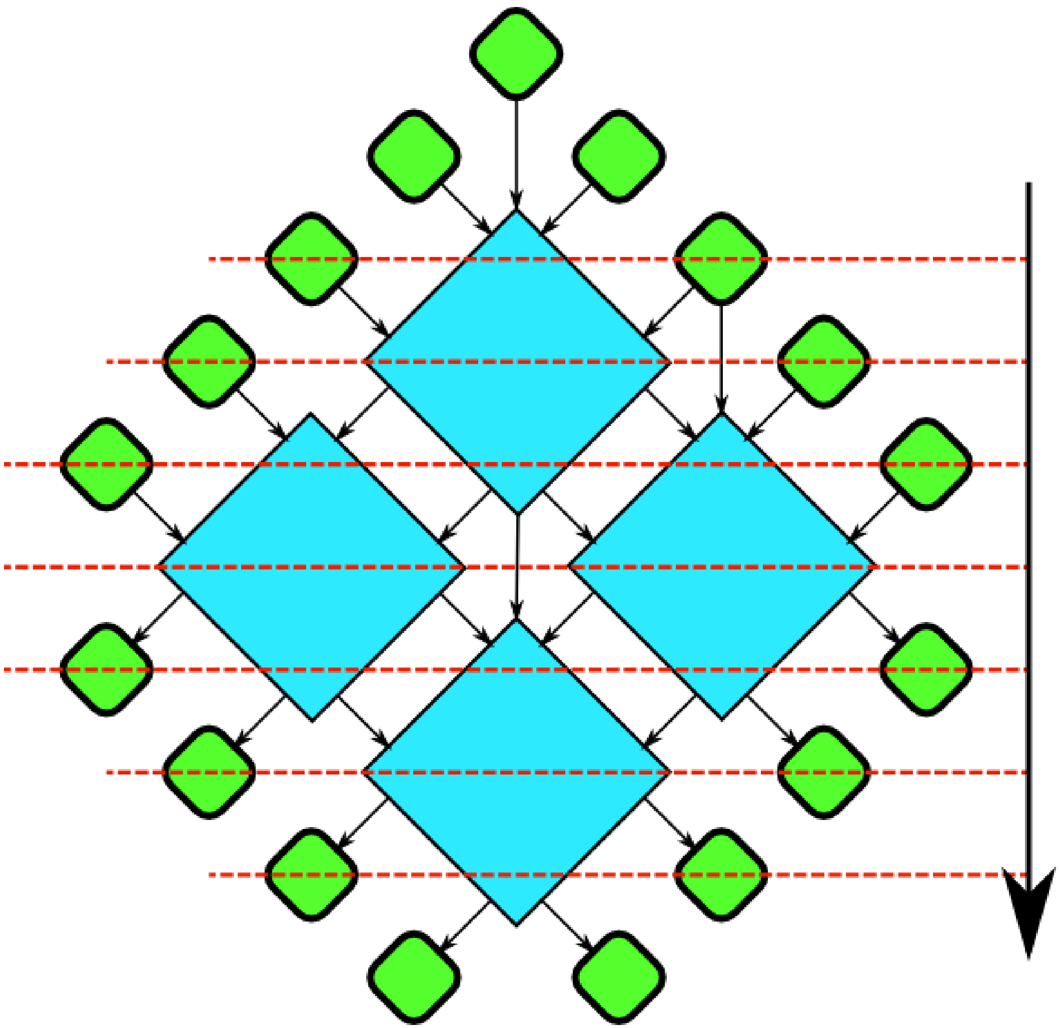
- Rotate recurrence to see sweep more clearly

Tiled Recurrence



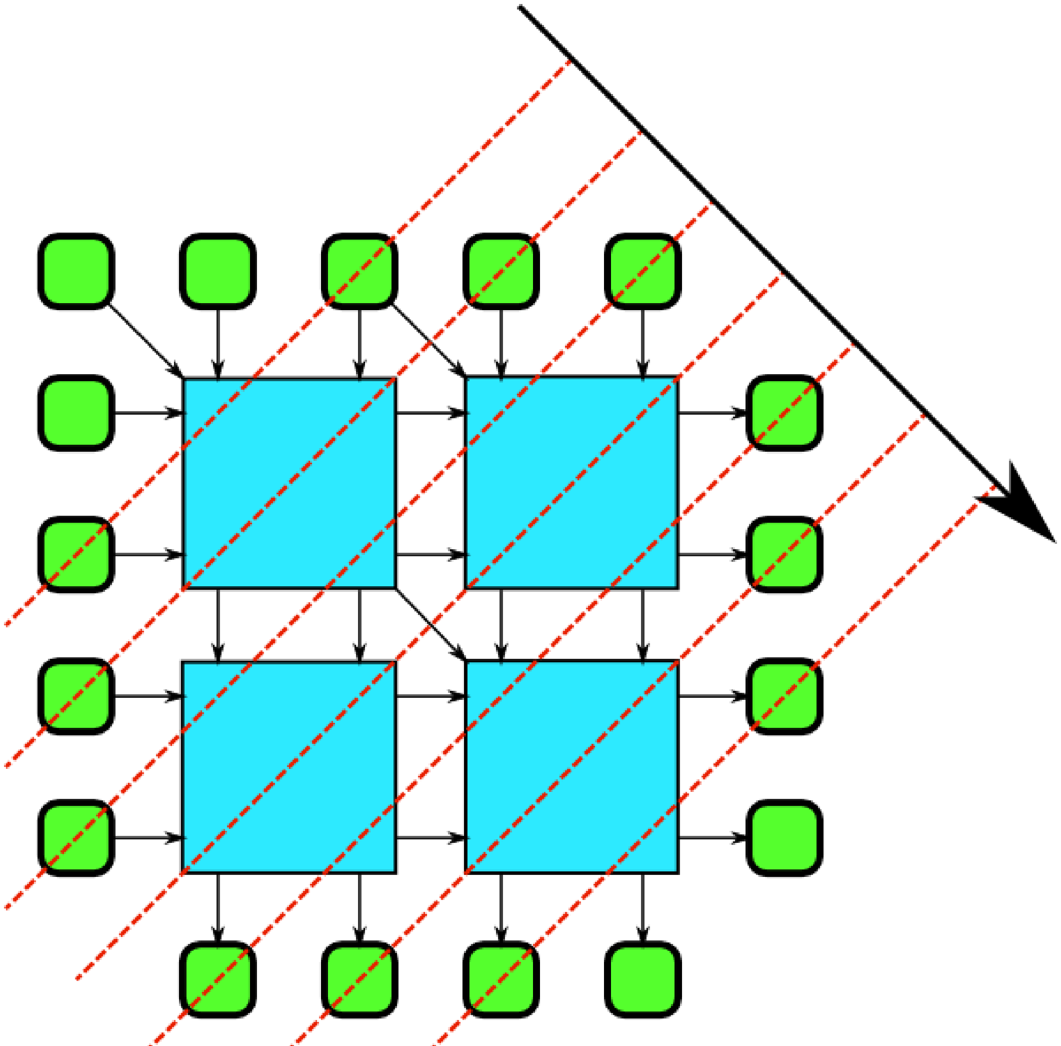
- Can partition recurrence to get a better compute vs. bandwidth ratio
- Show diamonds here, could also use paired trapezoids

Tiled Recurrence



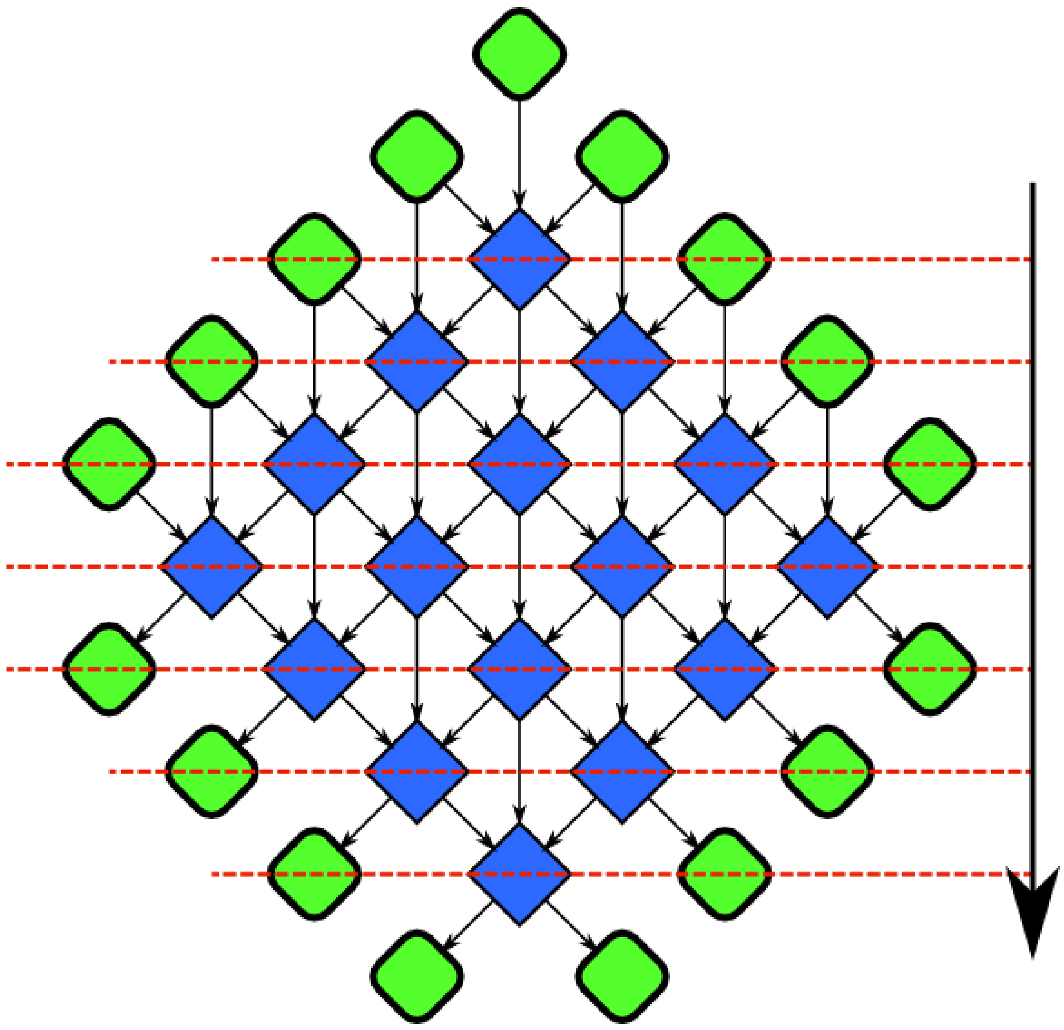
- Remove all non-redundant data dependences

Recursively Tiled Recurrences



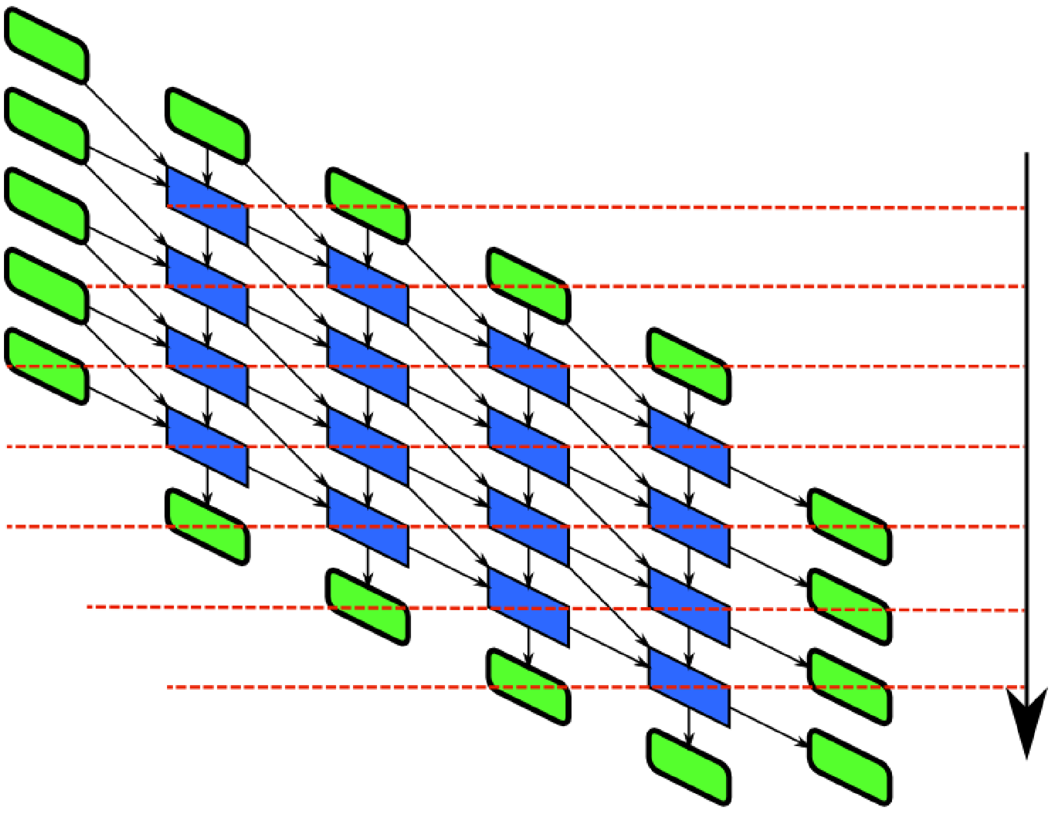
- Rotate back: same recurrence at a different scale!
- Leads to recursive cache-oblivious divide-and-conquer algorithm
- Implement with fork-join.

Rotated Recurrence



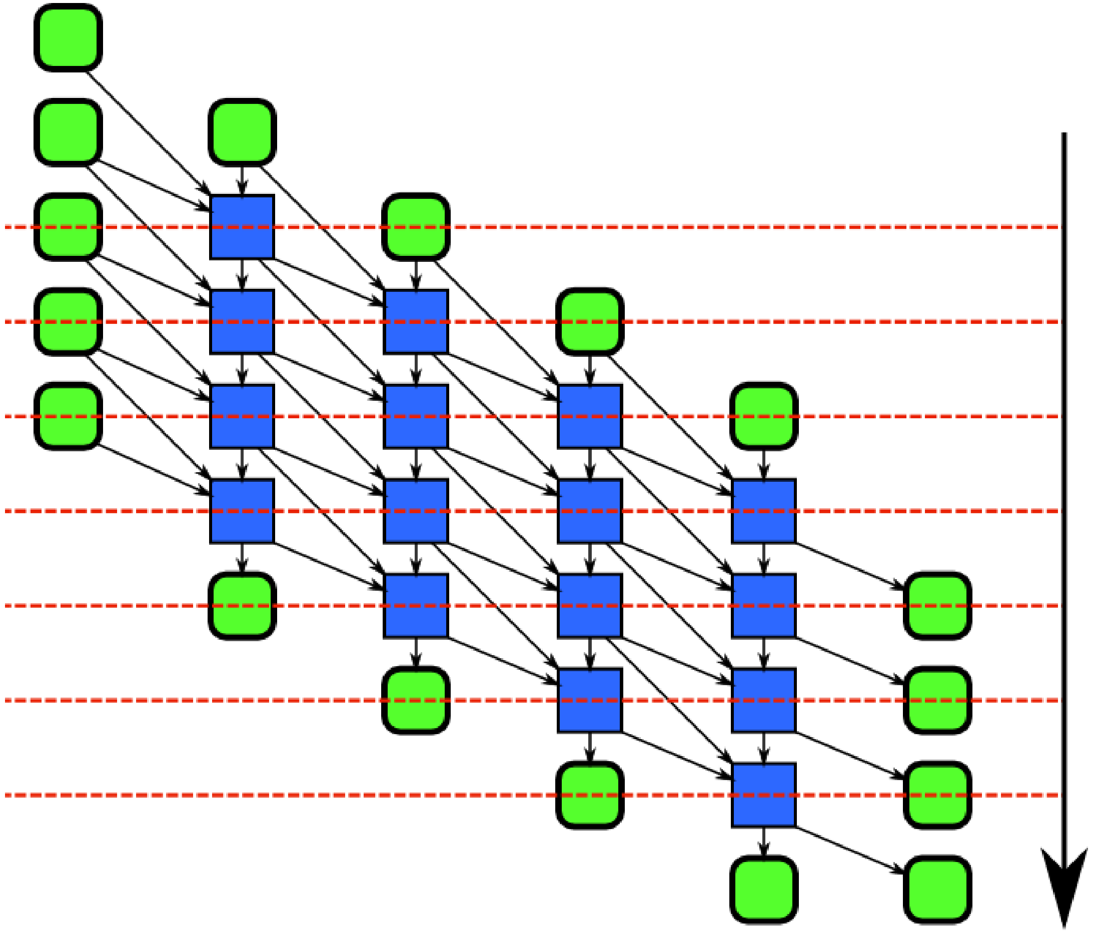
- Look at rotated recurrence again
- Let’s skew this by 45 degrees…

Skewed Recurrence



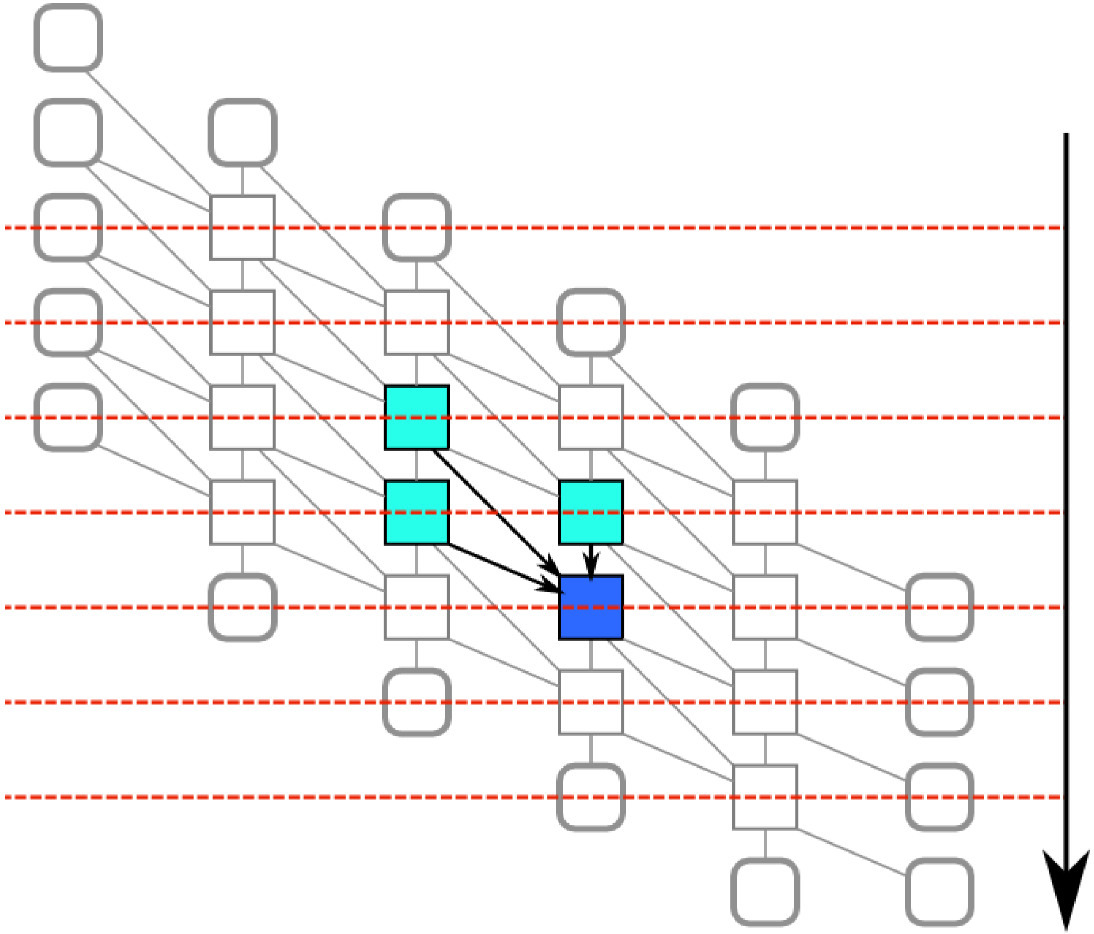
- A little hard to understand
- Let's just clean up the diagram a little bit…
  - Straighten up the symbols
  - Leave the data dependences as they are

Skewed Recurrence



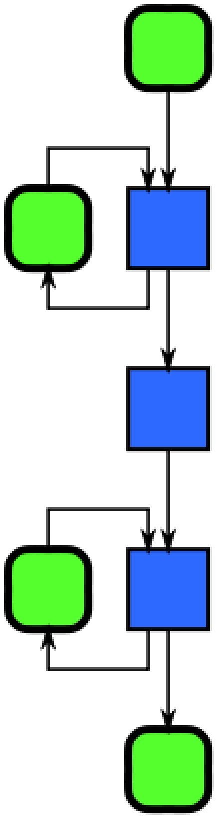
- This is a useful memory layout for implementing recurrences
- Let's now focus on one element
- Look at an element away from the boundaries

Recurrence = Iterated Stencil



- Each element depends on certain others in previous iterations
- An iterated stencil!
- *Convert iterated stencils into tiled recurrences for efficient implementation*

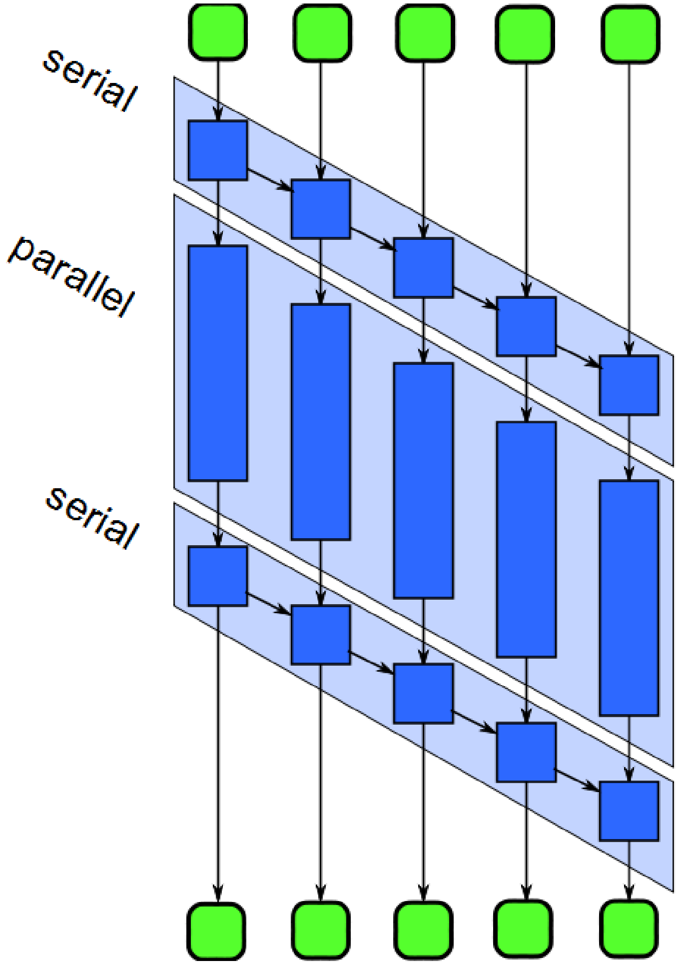
Pipeline



- *Pipeline* uses a sequence of stages that transform a flow of data
- Some stages may retain state
- Data can be consumed and produced incrementally: “online”

Examples: image filtering, data compression and decompression, signal processing

Pipeline



- Parallelize pipeline by
  - Running different stages in parallel
  - Running *multiple copies* of stateless stages in parallel

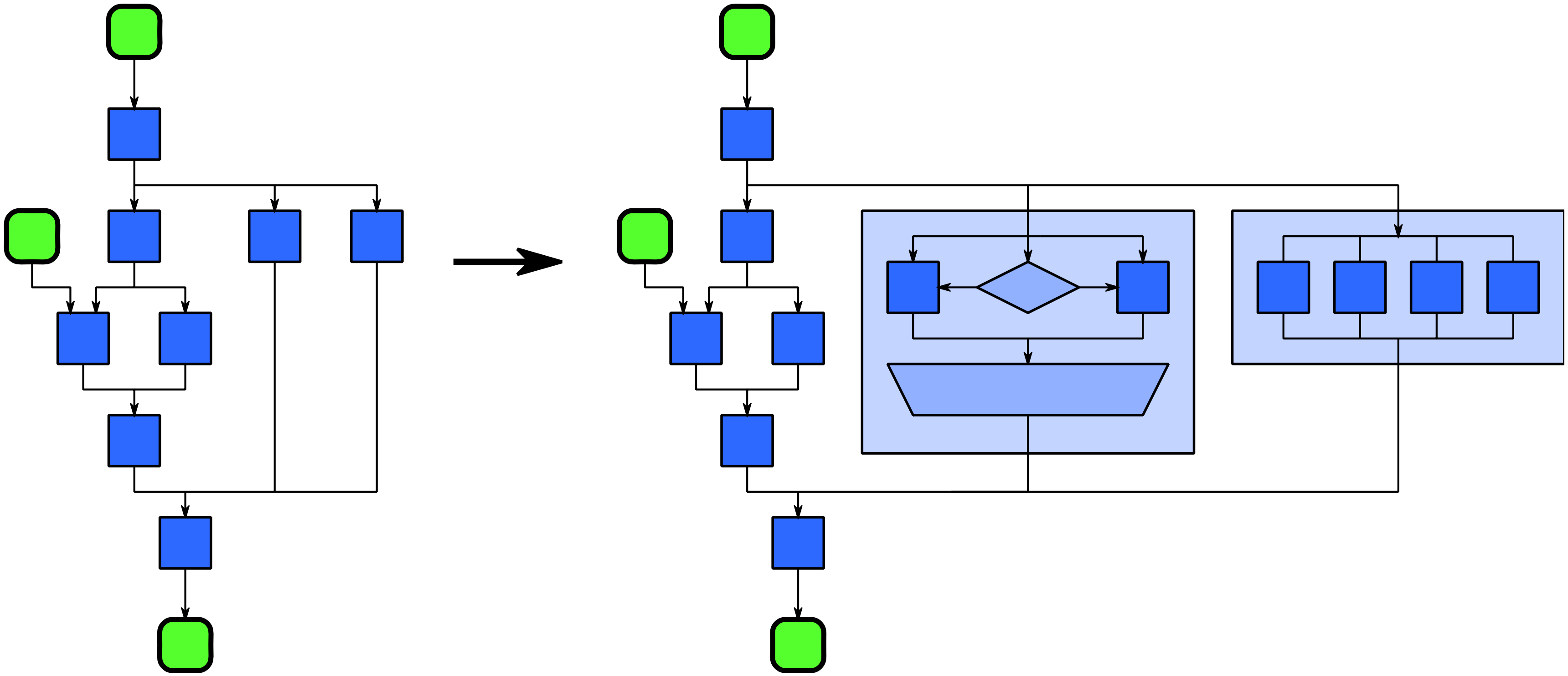
- Running multiple copies of stateless stages in parallel requires reordering of outputs

- Need to manage buffering between stages

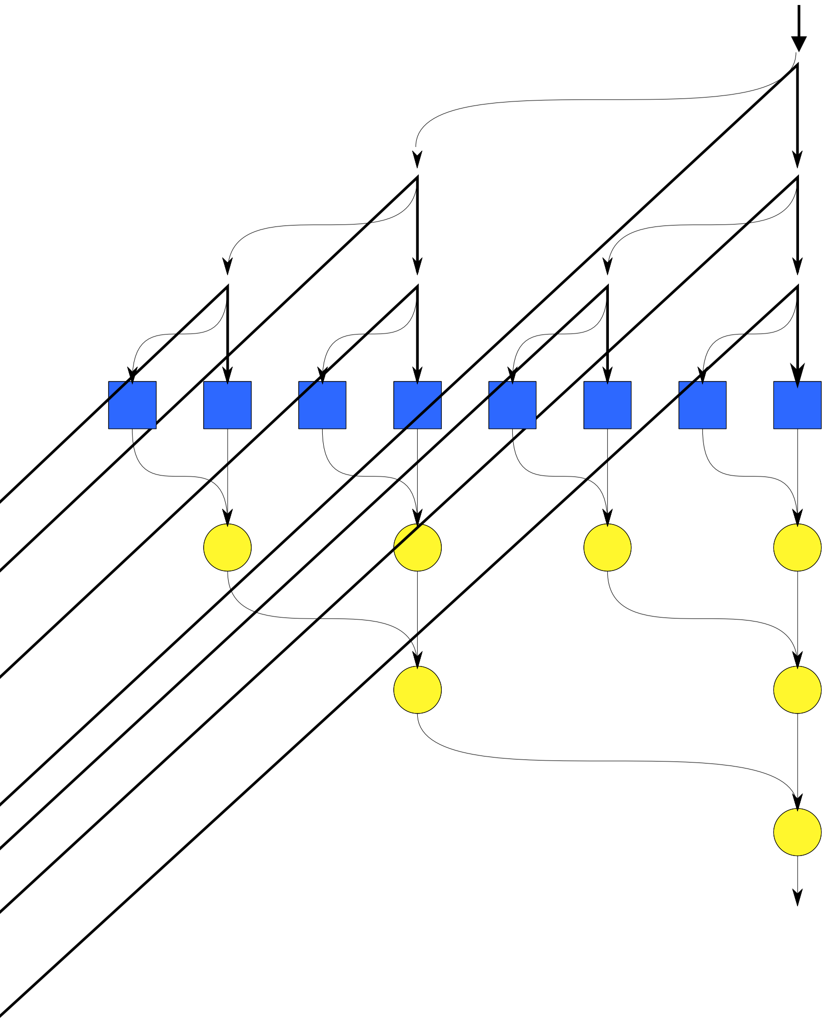
Recursive Patterns

- Recursion is an important "universal" serial pattern
  - Recursion leads to functional programming
  - Iteration leads to procedural programming
- Structural recursion: nesting of components
- Dynamic recursion: nesting of behaviors

Nesting: Recursive Composition



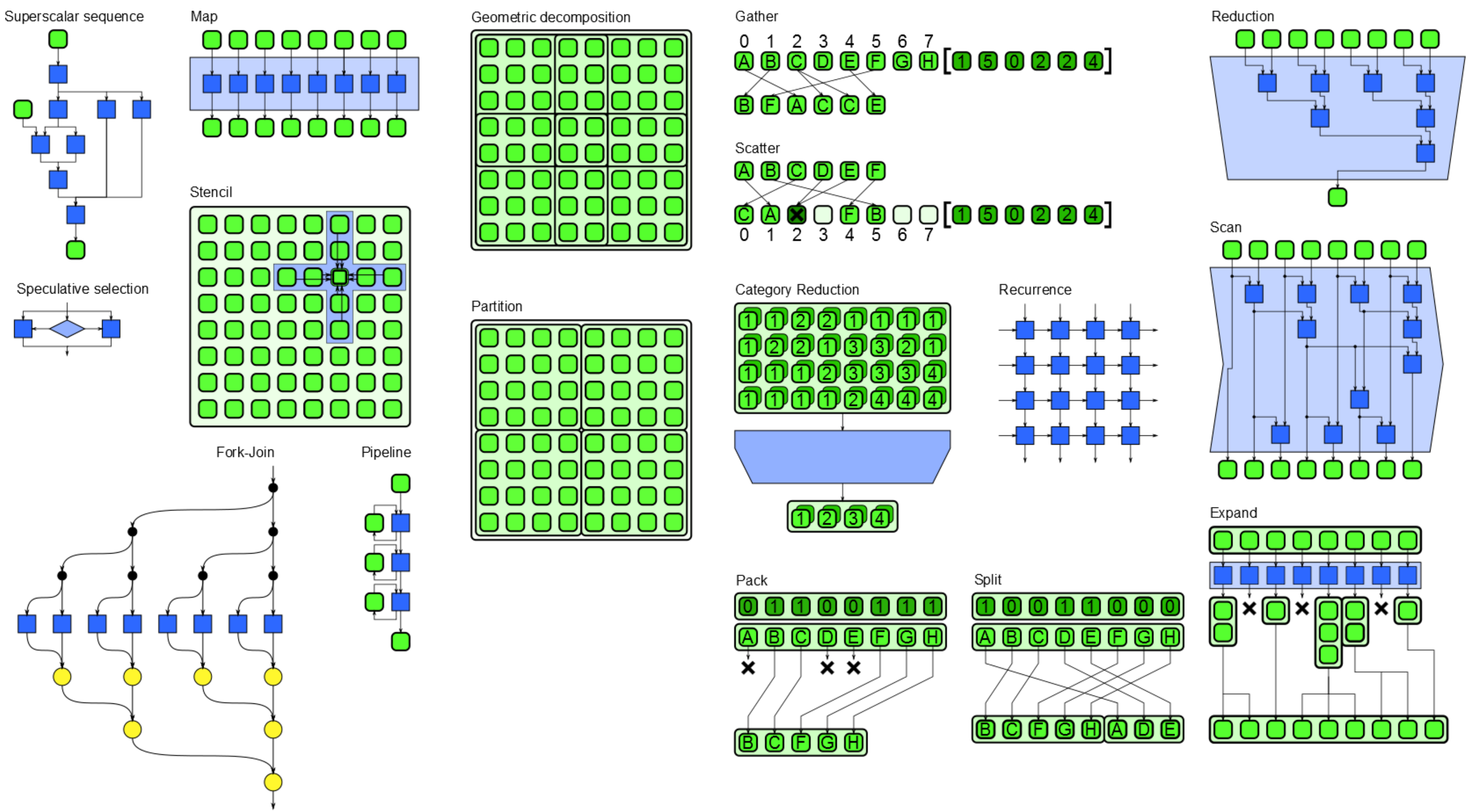
Fork-Join: Efficient Nesting



- Fork-join can be nested
- Spreads cost of work distribution and synchronization.
- This is how **cilk_for**, **tbb::parallel_for** and **arbb::map** are implemented.

Recursive fork-join enables high parallelism.

Parallel Patterns: Overview



Semantics and Implementation

Semantics: *What*

- The intended meaning as seen from the "outside"
- For example, for scan: compute all partial reductions given an associative operator

Implementation: *How*

- How it executes in practice, as seen from the "inside"
- For example, for scan: partition, serial reduction in each partition, scan of reductions, serial scan in each partition.
- *Many implementations may be possible*
- Parallelization may require reordering of operations
- Patterns should not over-constrain the ordering; only the important ordering constraints are specified in the semantics
- Patterns may also specify additional constraints, i.e. associativity of operators

Class students were given access to a cluster to work on for a week.

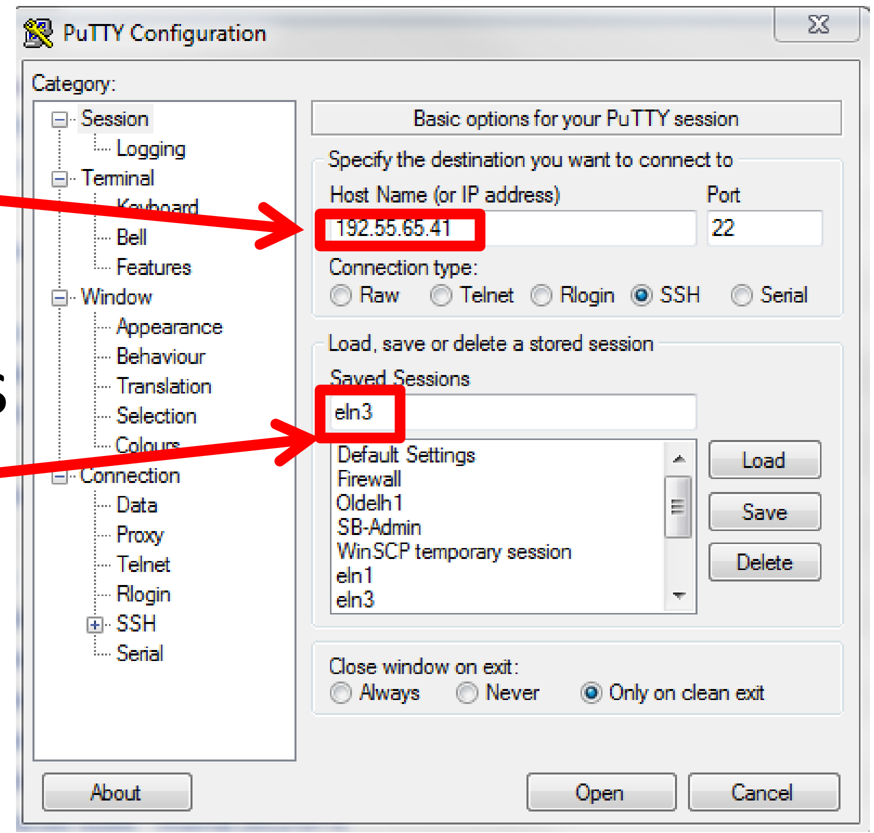
CLUSTER ACCESS

Running Examples on Endeavour

- Connect via SSH or Putty
- (optional) use “screen” to protect against disconnects
- Connect to a compute node in an LSF job
- Set up compiler environment
- Compile and run on a host system
- Compile and run on a Intel® Xeon Phi™ coprocessor

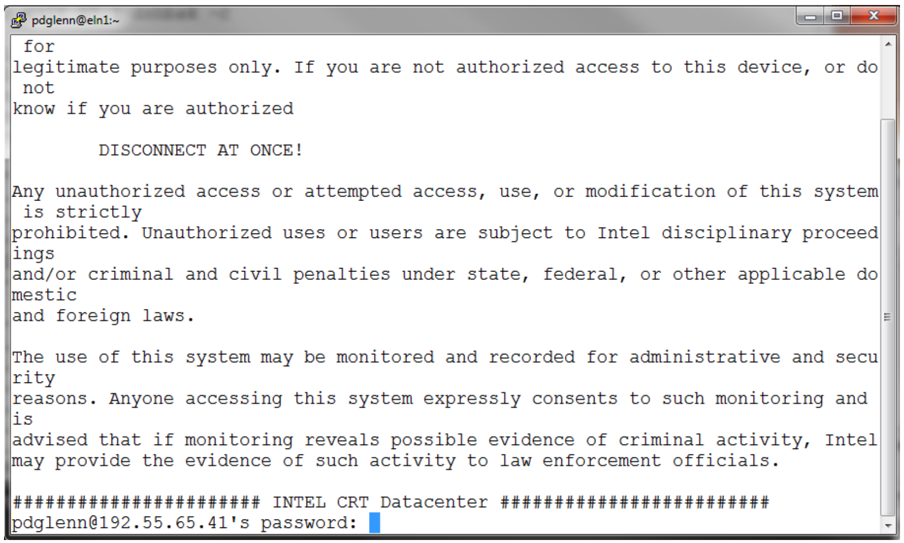
Connecting with PuTTY

- Under Session Fill in IP address: 207.108.8.212
- Be sure to give this entry descriptive name such as Endeavor.
- Click “save”
- Click “open”



Connecting with PuTTY II

- A successful connection should look like:



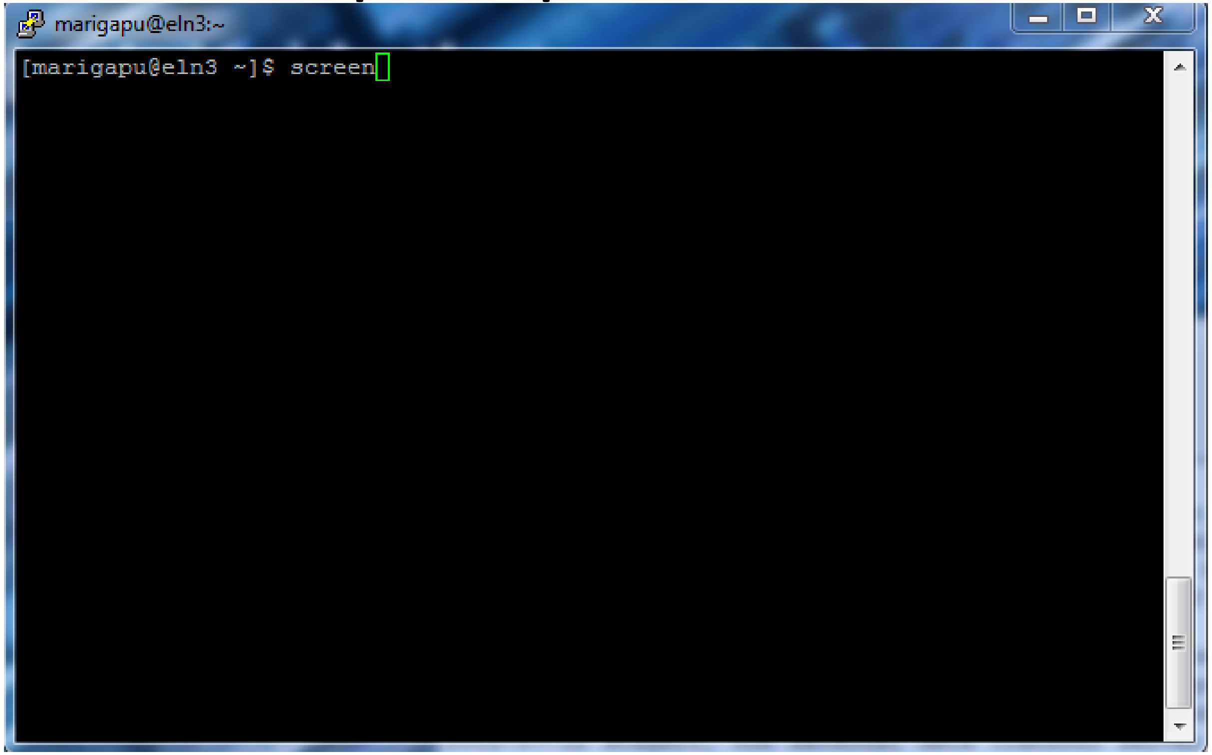
- Enter user name and password

Introduction to “screen”

- It is a screen manager with VT100/ANSI terminal emulation
- Screen is a full-screen window manager that multiplexes a physical terminal between several processes (typically interactive shells)
- Allows to host multiple parallel programs in a single Putty window
- Protects from connection resets - allows you to reconnect after you establish a new ssh connection
- All windows run their programs completely independent of each other. Programs continue to run when their window is currently not visible and even when the whole screen session is detached from the user’s terminal.
- when a program terminates, screen (per default) kills the Window that contained it. If this window was in the foreground, the display switches to the previous window;
- if none are left, screen exits.

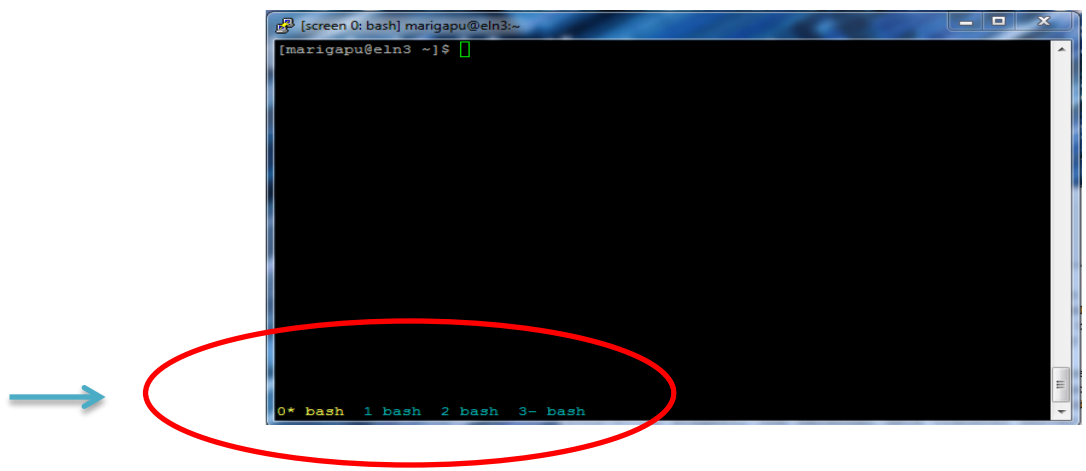
Screen Example I

- Logon to Endeavour login node as usual. Type 'screen' at the prompt.



Screen Example II

- By simply pressing “Ctrl-a c” keys, you can open another shell window within the same screen session. At the bottom you will see the session number. You can switch between the sessions by pressing “Ctrl-a <number_of_the screen>”



Screen Example III

- You can recover the shell back by doing ‘screen -r’. You will be able to get the same shell where you left off. You can end the screen session by either ‘exit’ or ‘Ctrl d’

Start an Interactive LSF Job

- Reserve a compute node for 600 minutes:

```
$ bsub -R 1*"{select[ivt]}" -W 600
  -Is /bin/bash
```

- Wait till you see:
  `Prologue finished, program starts`
- Retrieve hostname of the system

```
$ hostname
esg061
```

- Exit the job:

```
$ logout
```

Compile and Run on HOST

- Reserve a node

```
$ bsub -R '1*{select[ivt]}' -W 600
   -Is /bin/bash
```

- Source compiler environment

```
$ . env.sh
```

- Compile and run your program

```
$ icc –o foo foo.c
$ ./foo
$ icpc –o foopp foo.cpp
$ ./foopp
```

Compile and Run on Intel®Xeon Phi™

- Source compiler environment

```
$ . env.sh
```

- Compile your program

```
$ icc –o foo –mmic foo.c
```

- ssh to the Intel® Xeon Phi™ coprocessor and execute your program

```
$ ssh `hostname`-mic0
$ . env-mic.sh
$ ./foo
```

PROGRAMMING MODELS

Intel’s Parallel Programming Models

Intel® Cilk™ Plus	Intel® Threading Building Blocks	Domain Specific Libraries	Established Standards	Research and Development
C/C++ language extensions to simplify parallelism	Widely used C++ template library for parallelism	Intel® Integrated Performance Primitives Intel® Math Kernel Library	Message Passing Interface (MPI) OpenMP* Coarray Fortran OpenCL*	Intel® Concurrent Collections Offload Extensions Intel® Array Building Blocks River Trail: parallel javascript Intel® SPMD Parallel Compiler
Open sourced Also an Intel product	Open sourced Also an Intel product			

Choice of high-performance parallel programming models

- Libraries for pre-optimized and parallelized functionality
- Intel® Cilk™ Plus and Intel® Threading Building Blocks supports composable parallelization of a wide variety of applications.
- OpenCL* addresses the needs of customers in specific segments, and provides developers an additional choice to maximize their app performance
- MPI supports distributed computation, combines with other models on nodes

Intel's Parallel Programming Models

- ***Intel® Cilk™ Plus: Compiler extension***
  - Fork-join parallel programming model
  - Serial semantics if keywords are ignored (serial elision)
  - Efficient work-stealing load balancing, hyperobjects
  - Supports vector parallelism via array slices and elemental functions
- ***Intel® Threading Building Blocks (TBB): Library***
  - Template library for parallelism
  - Efficient work-stealing load balancing
  - Efficient low-level primitives (atomics, memory allocation).

SSE Intrinsics

Plain C/C++

```
float sprod(float *a,
            float *b,
            int size) {
  float sum = 0.0;
  for (int i=0; i < size; i++)
    sum += a[i] * b[i];
  return sum;
}
```

SSE

```
float sprod(float *a,
            float *b,
            int size){
  _declspec(align(16))
    __m128 sum, prd, ma, mb;
  float tmp = 0.0;
  sum = _mm_setzero_ps();
  for(int i=0; i<size; i+=4){
    ma = _mm_load_ps(&a[i]);
    mb = _mm_load_ps(&b[i]);
    prd = _mm_mul_ps(ma,mb);
    sum = _mm_add_ps(prd,sum);
  }
  prd = _mm_setzero_ps();
  sum = _mm_hadd_ps(sum, prd);
  sum = _mm_hadd_ps(sum, prd);
  _mm_store_ss(&tmp, sum);
  return tmp;
}
```

SSE Intrinsics

Plain C/C++

```
float sprod(float *a,
            float *b,
            int size) {
  float sum = 0.0;
  for (int i=0; i < size; i++)
    sum += a[i] * b[i];
  return sum;
}
```

Problems with SSE code:

- Machine dependent
    - Assumes vector length 4
- Verbose
- Hard to maintain
- Only vectorizes
    - SIMD instructions, no threads
- Example not even complete:
    - Array must be multiple of vector length

SSE

```
float sprod(float *a,
            float *b,
            int size){
  _declspec(align(16))
    __m128 sum, prd, ma, mb;
  float tmp = 0.0;
  sum = _mm_setzero_ps();
  for(int i=0; i<size; i+=4){
    ma = _mm_load_ps(&a[i]);
    mb = _mm_load_ps(&b[i]);
    prd = _mm_mul_ps(ma,mb);
    sum = _mm_add_ps(prd,sum);
  }
  prd = _mm_setzero_ps();
  sum = _mm_hadd_ps(sum, prd);
  sum = _mm_hadd_ps(sum, prd);
  _mm_store_ss(&tmp, sum);
  return tmp;
}
```

Cilk™ Plus

Plain C/C++

```
float sprod(float *a,
            float *b,
            int size) {
  float sum = 0;
  for (int i=0; i < size; i++)
    sum += a[i] * b[i];
  return sum;
}
```

Cilk™ Plus

```
float sprod(float a*,
            float b*,
            int size) {
  return __sec_reduce_add(
    a[0:size] * b[0:size] );
}
```

Cilk™ Plus + Partitioning

Plain C/C++

```
float sprod(float *a,
            float *b,
            int size) {
  float sum = 0;
  for (int i=0; i < size; i++)
    sum += a[i] * b[i];
  return sum;
}
```

Cilk™ Plus

```
float sprod(float* a,
            float* b,
            int size) {
  int s = 4096;
  cilk::reducer<cilk::op_add<float> >sum(0);
  cilk_for (int i=0; i<size; i+=s) {
    int m = std::min(s,size-i);
    *sum += __sec_reduce_add(
      a[i:m] * b[i:m] );
  }
  return sum.get_value();
}
```

TBB

Plain C/C++

```
float sprod(float *a,
            float *b,
            int size) {
  float sum = 0;
  for (int i=0; i < size; i++)
    sum += a[i] * b[i];
  return sum;
}
```

TBB

```
float sprod(const float a[],
            const float b[],
            size_t n ) {
  return tbb::parallel_reduce(
    tbb::blocked_range<size_t>(0,n),
    0.0f,

    [=](
      tbb::blocked_range<size_t>& r,
      float in
    ) {
      return std::inner_product(
        a+r.begin(), a+r.end(),
        b+r.begin(), in );
      },
      std::plus<float>()
  );
}
```

Patterns in Intel's Parallel Programming Models

Intel® Cilk™ Plus

- cilk_spawn, cilk_sync: nesting, fork-join
- Hyperobjects: reduce
- cilk_for, elemental functions: map
- Array notation: scatter, gather

Intel® Threading Building Blocks

- parallel_invoke, task_group: nesting, fork-join
- parallel_for, parallel_foreach: map
- parallel_do: workpile (map + incr. task addition)
- parallel_reduce, parallel_scan: reduce, scan
- parallel_pipeline: pipeline
- flow_graph: plumbing for reactive and streaming

Conclusions

- **Explicit parallelism is a *requirement* for scaling**
  - Moore's Law is still in force.
  - However, it is about number of transistors on a chip, not scalar performance.
- **Patterns are a structured way to think about applications and programming models**
  - Useful for communicating and understanding structure
  - Useful for achieving a scalable implementation
- **Good parallel programming models support scalable parallel patterns**
  - Parallelism, data locality, determinism
  - Low-overhead implementations

MACHINE MODELS

Course Outline

- Introduction
  - Motivation, goals, patterns
- **Background**
  - **Machine model, complexity, work-span**
- Cilk™ Plus and Threading Building Blocks
  - Programming model
  - Examples
- Practical matters
  - Debugging and profiling

Background: Outline

- Machine model
  - Parallel hardware mechanisms
  - Memory architecture and hierarchy
- Speedup and efficiency
- DAG model of computation
  - Greedy scheduling
- Work-span parallel complexity model
  - Brent's Lemma and Amdahl's Law
  - Amdahl was an optimist: better bounds with work-span
  - Parallel slack
  - Potential vs. actual parallelism

What you (probably) want

Performance

- Compute results efficiently
- Improve absolute computation times over serial implementations

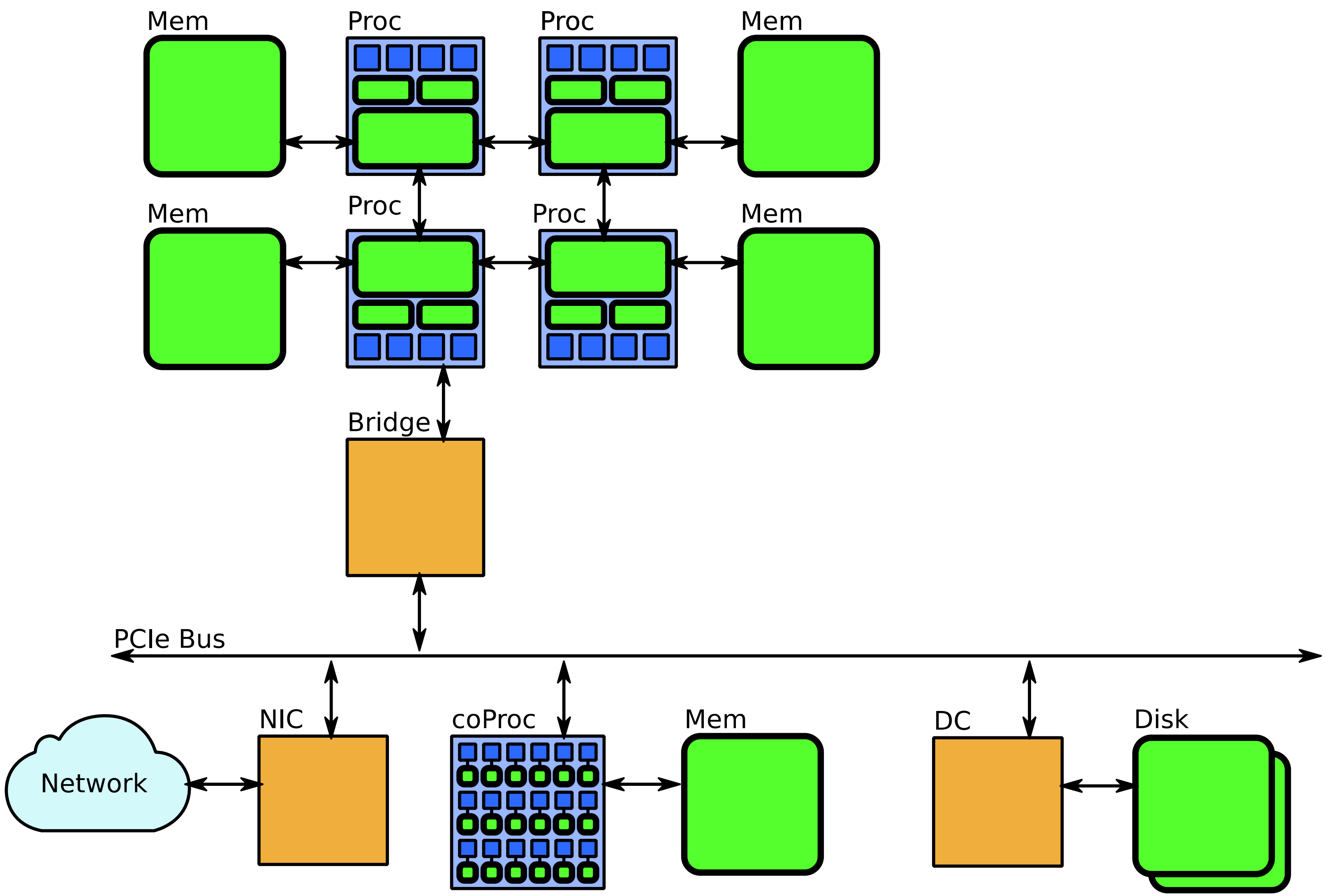
Portability

- Code that work well on a variety of machines without significant changes
- Scalable: make efficient use of more and more cores

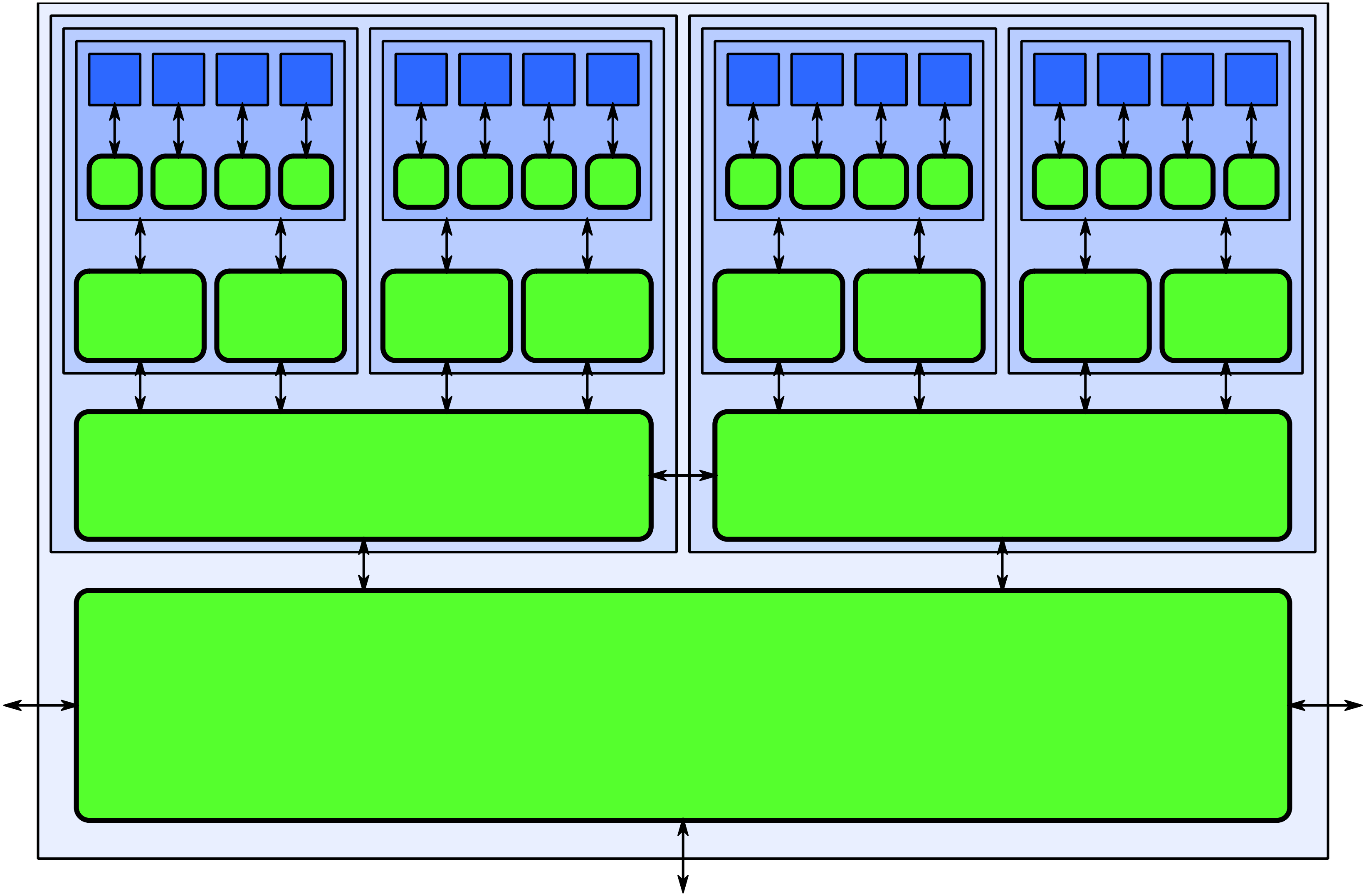
Productivity

- Write code in a short period of time
- Debug, validate, and maintain it efficiently

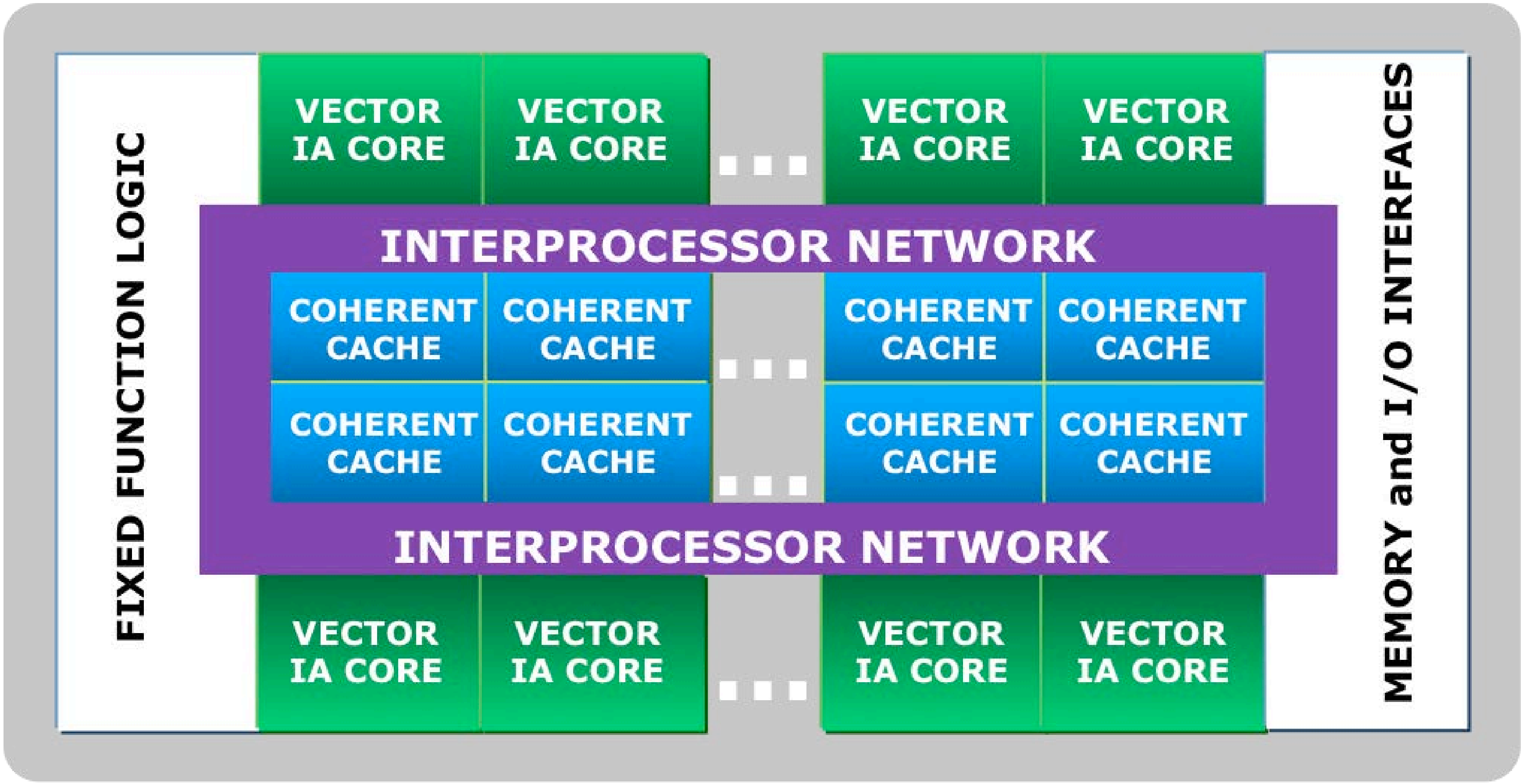
Typical System Architecture



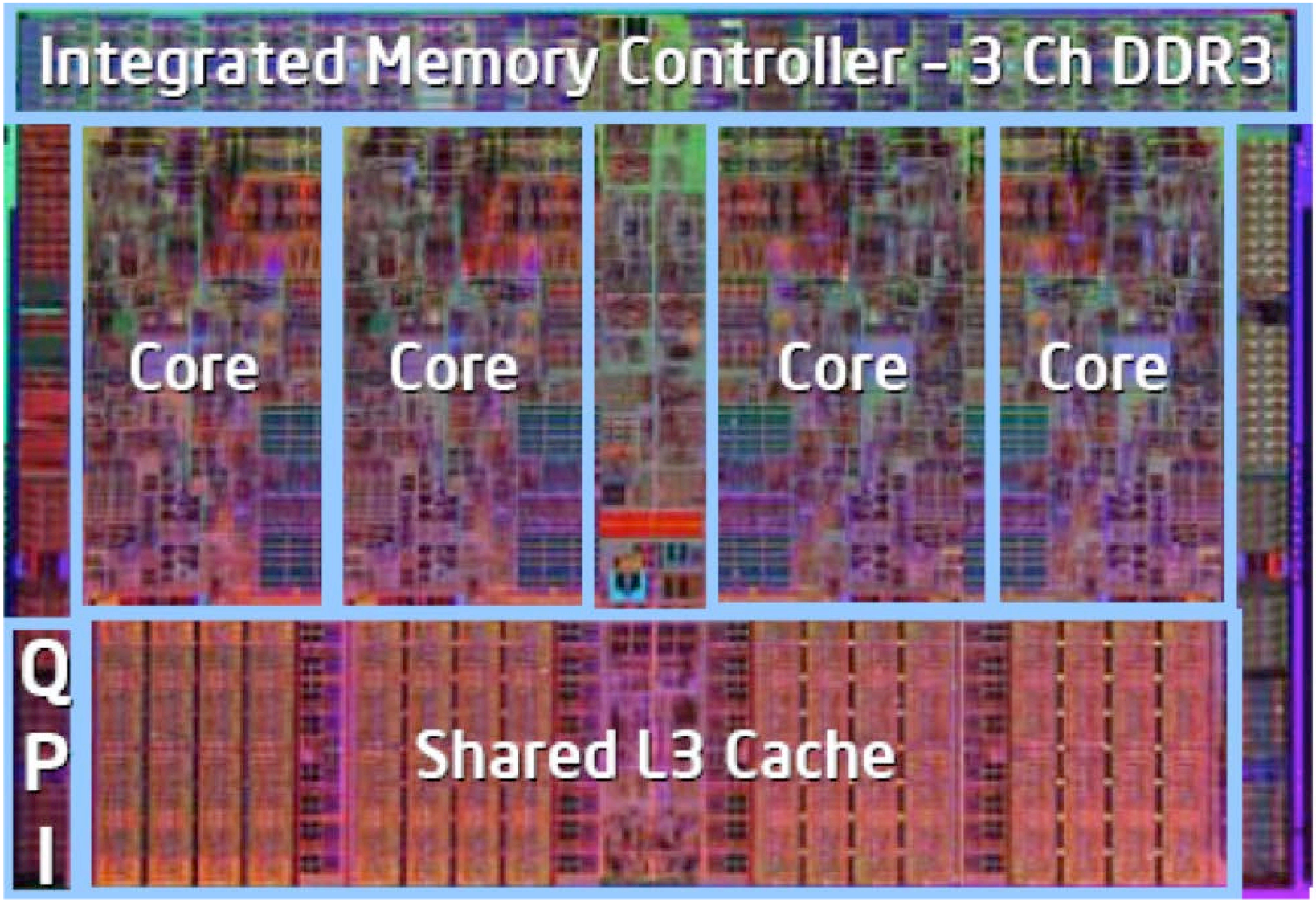
Cache Hierarchy



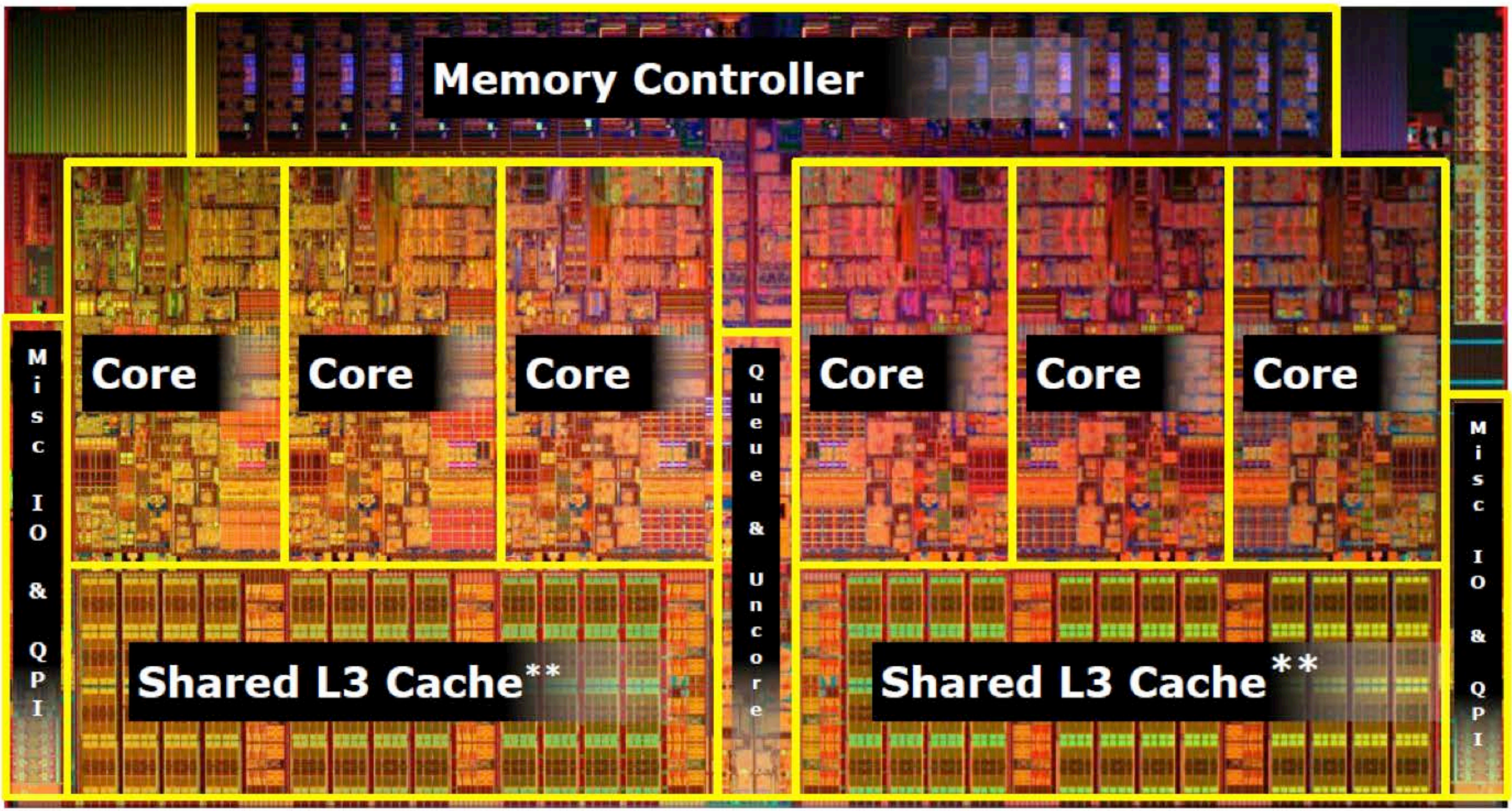
Xeon Phi (MIC) Architecture



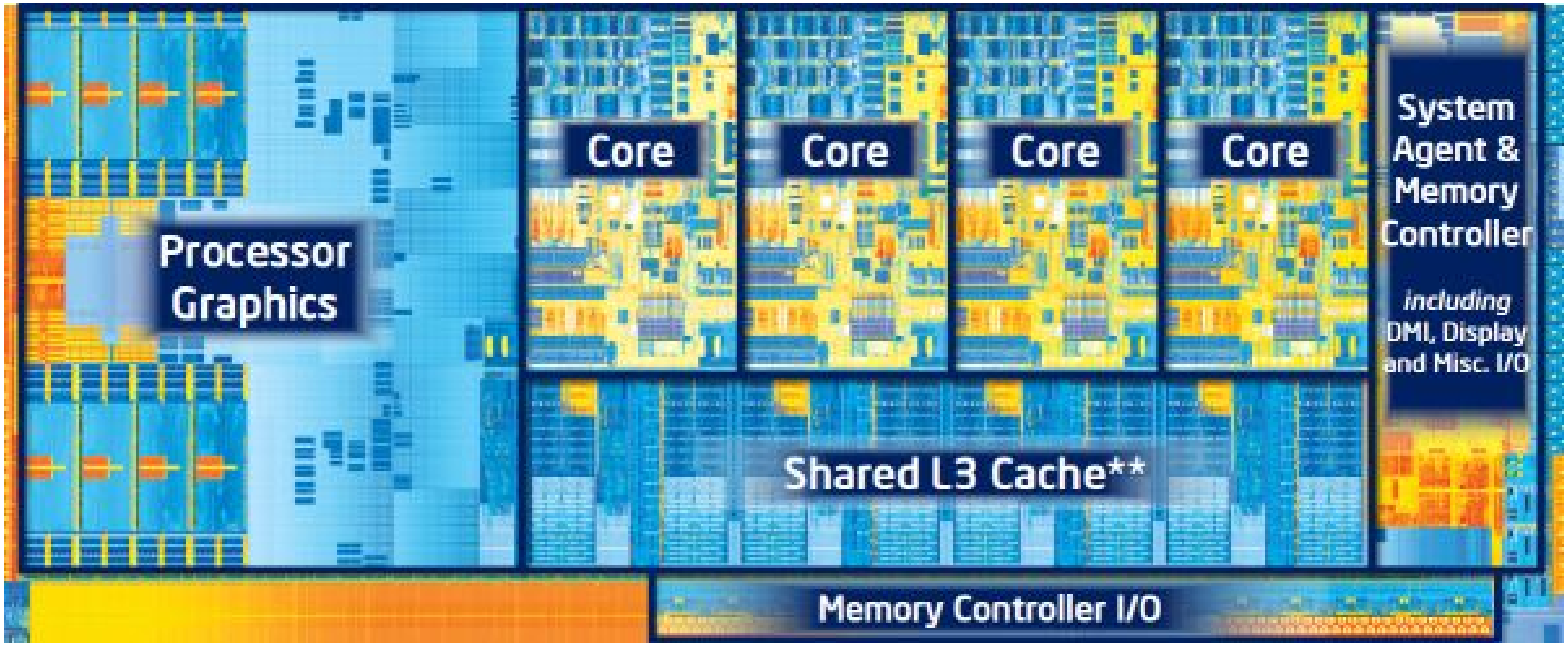
Nehalem


Integrated Memory Controller - 3 Ch DDR3
Core
Core
Core
Core
Q
P
I
Shared L3 Cache

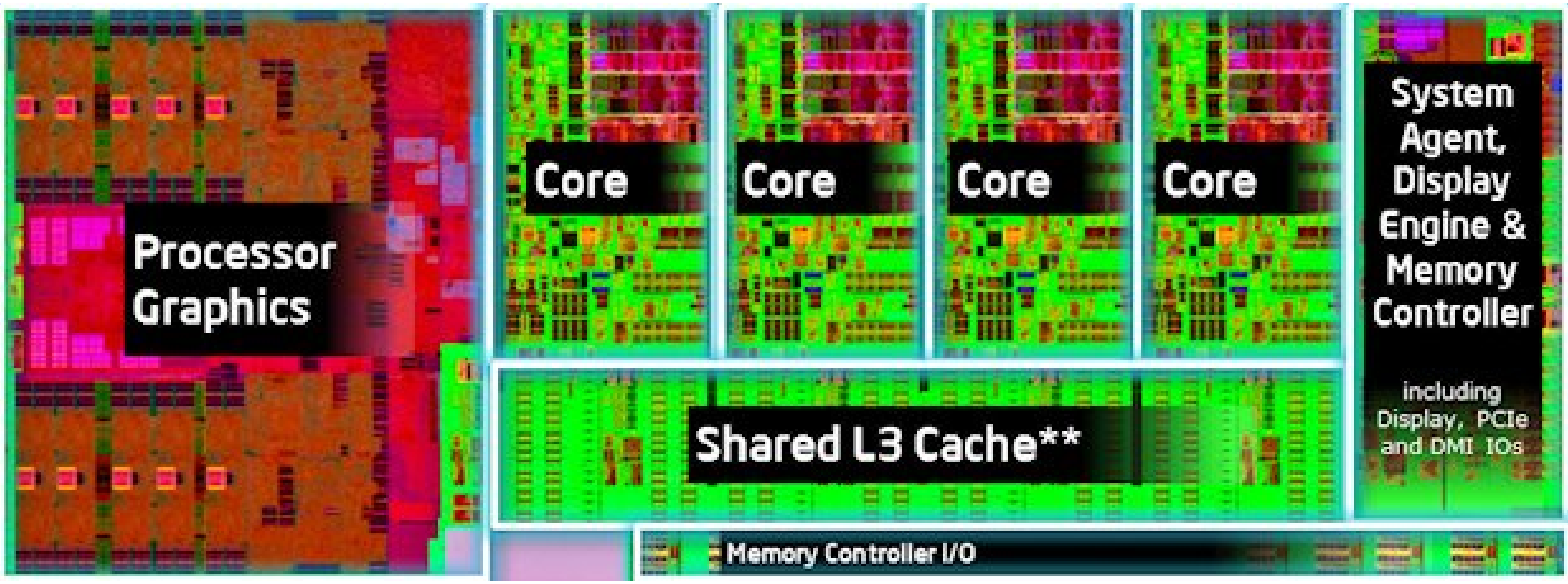
Westmere


Memory Controller
Misc IO & QPI
Core
Core
Core
Queue & Uncore
Core
Core
Core
Misc IO & QPI
Shared L3 Cache**
Shared L3 Cache**

Ivy Bridge


Processor Graphics
Core
Core
Core
Core
System Agent & Memory Controller
including DMI, Display and Misc. I/O
Shared L3 Cache**
Memory Controller I/O

Haswell



Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O

Key Factors

Compute: Parallelism

What mechanisms do processors provide for using parallelism?

- Implicit: instruction pipelines, superscalar issues
- Explicit: cores, hyperthreads, vector units

- *How to map potential parallelism to actual parallelism?*

Data: Locality

How is data managed and accessed, and what are the performance implications?

- Cache behavior, conflicts, sharing, coherency, (false) sharing; alignments with cache lines, pages, vector lanes

- *How to design algorithms that have good data locality?*

Pitfalls

Load imbalance

- Too much work on some processors, too little on others

Overhead

- Too little real work getting done, too much time spent managing the work

Deadlocks

- Resource allocation loops causing lockup

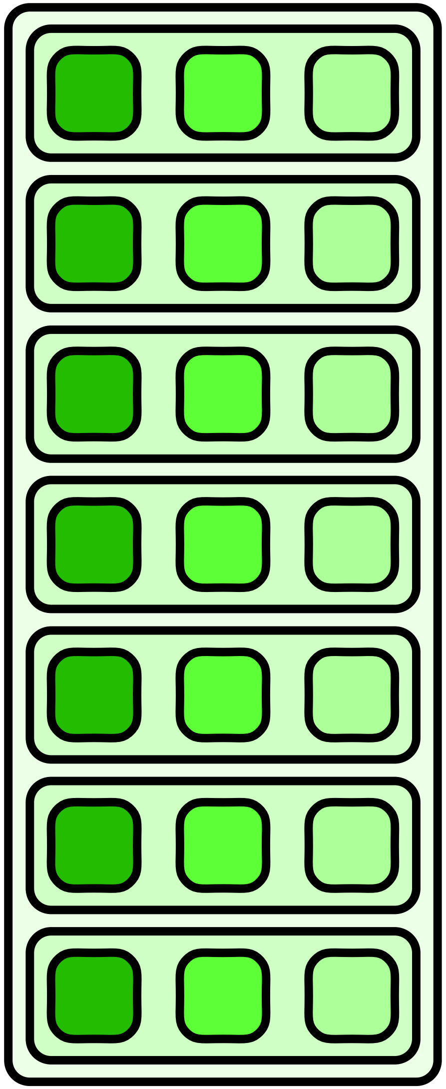
Race conditions

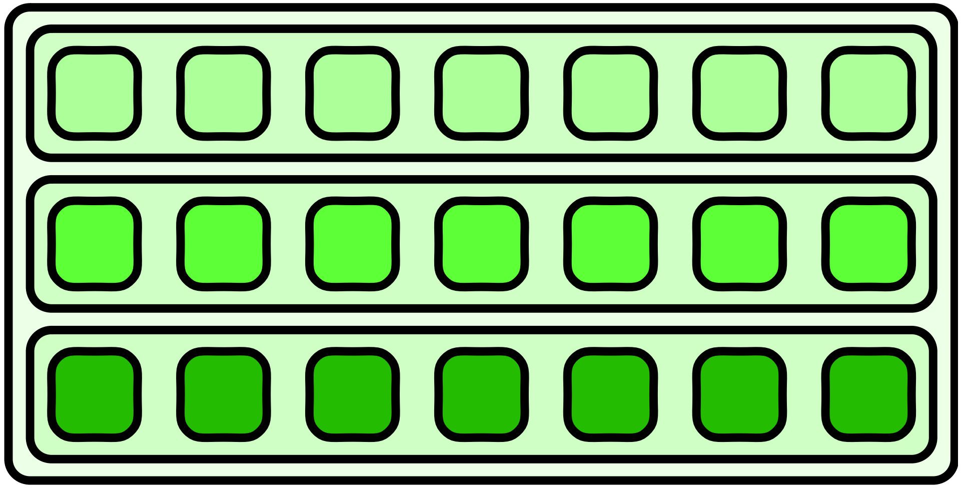
- Incorrect interleavings permitted, resulting in incorrect and non-deterministic results

Strangled scaling

- Contended locks causing serialization

Data Layout: AoS vs. SoA



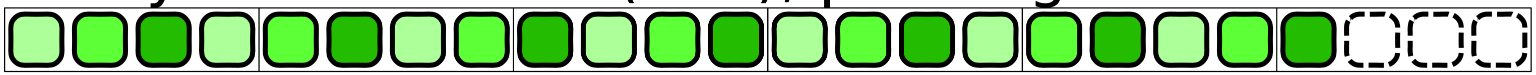


Array of structures (AoS) tends to cause cache alignment problems, and is hard to vectorize.

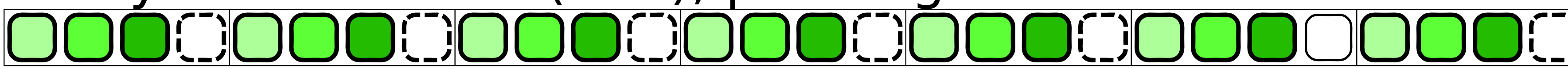
Structure of arrays (SoA) can be easily aligned to cache boundaries and is vectorizable.

Data Layout: Alignment

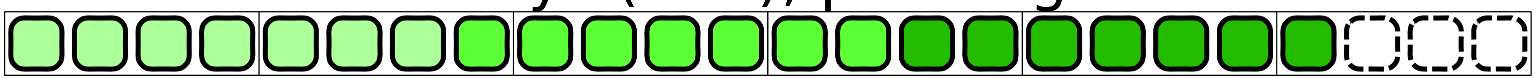
Array of Structures (AoS), padding at end.



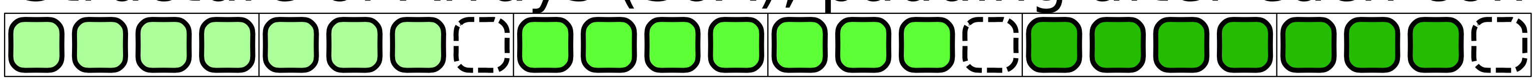
Array of Structures (AoS), padding after each structure.



Structure of Arrays (SoA), padding at end.



Structure of Arrays (SoA), padding after each component.



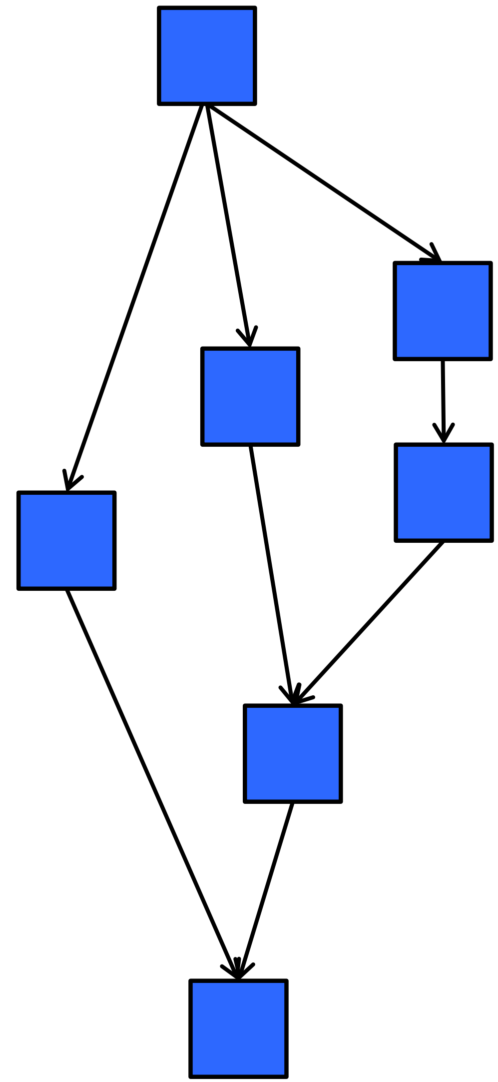
COMPLEXITY MEASURES

Speedup and Efficiency

- T_1 = time to run with 1 worker
- T_P = time to run with P workers
- T_1/T_P = *speedup*
  - The relative reduction in time to complete the same task
  - Ideal case is linear in P
    - i.e. 4 workers gives a best-case speedup of 4.
  - In real cases, speedup often significantly less
  - In *rare* cases, such as search, *can* be superlinear
- $T_1/(PT_P)$ = *efficiency*
  - 1 is perfect efficiency
  - Like linear speedup, perfect efficiency is hard to achieve
  - Note that this is not the same as "utilization"

DAG Model of Computation

- Program is a directed acyclic graph (DAG) of tasks
- The hardware consists of workers
- Scheduling is *greedy*
  - No worker idles while there is a task available.



Departures from Greedy Scheduling

- Contended mutexes.
  – Blocked worker could be doing another task

Avoid mutexes, use wait-free atomics instead.

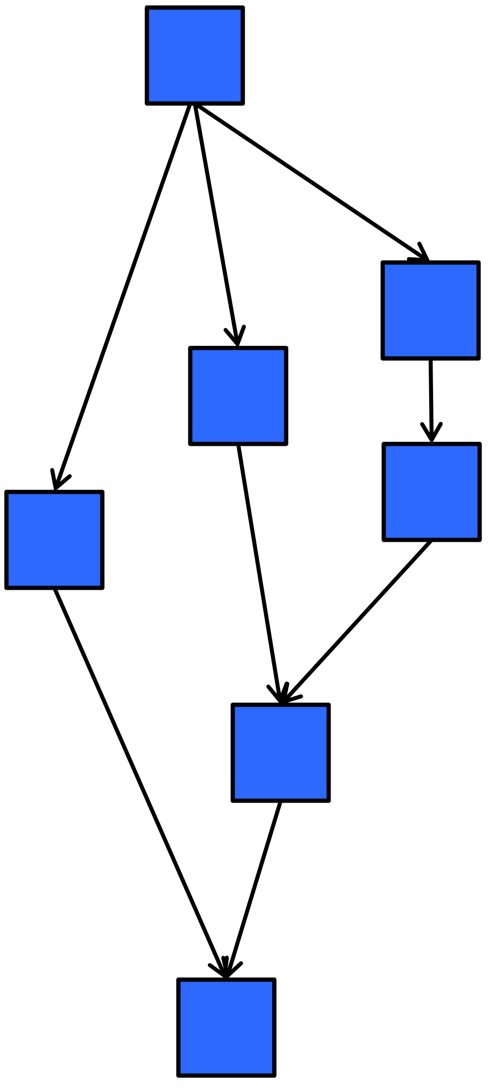
- One linear stack per worker
  – Caller blocked until callee completes

Intel® Cilk™ Plus has cactus stack.

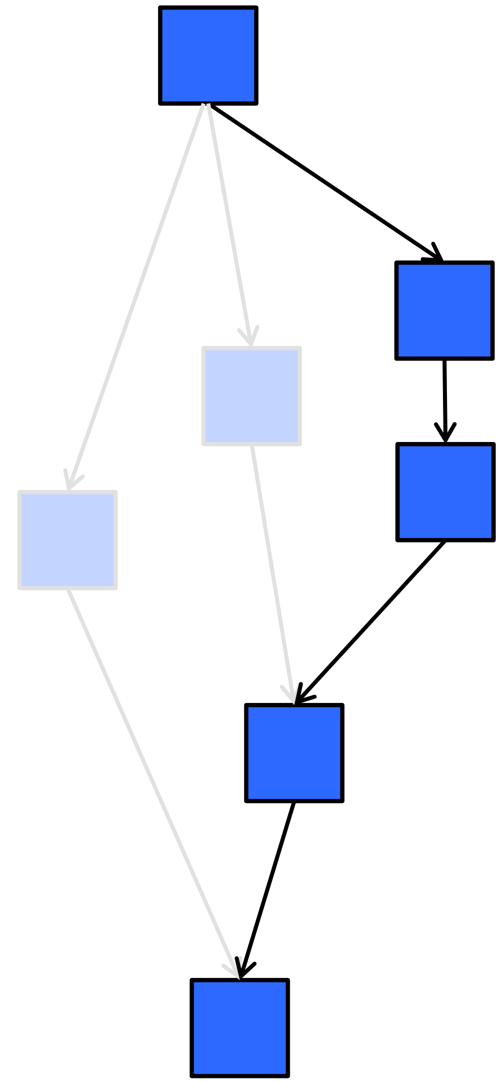
Intel® TBB uses continuation-passing style inside algorithm templates.

Work-Span Model

- T_P = time to run with P workers
- T_1 = *work*
  - time for serial execution
  - sum of all work
- T_∞ = *span*
  - time for *critical path*



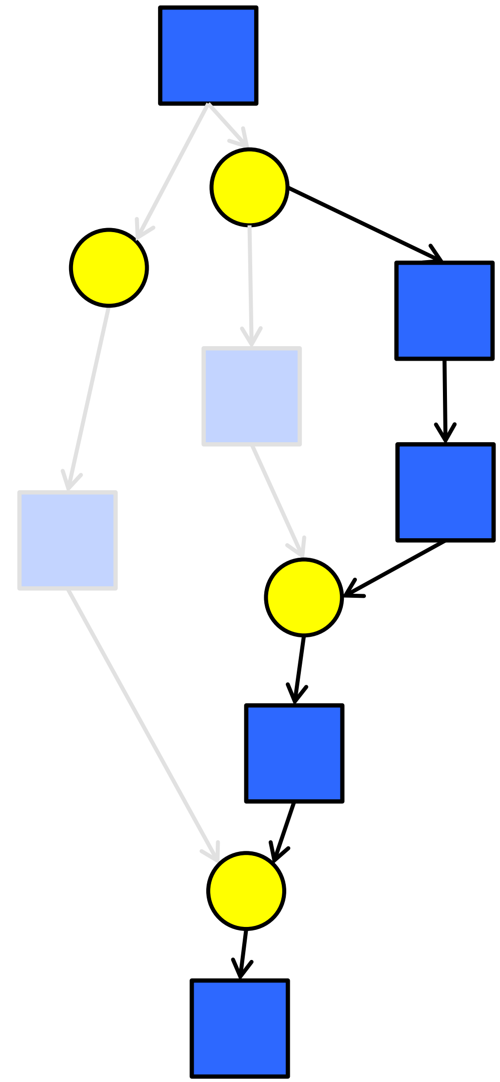
Work-Span Example



T_1 = *work* = 7
T_∞ = *span* = 5

Burdened Span

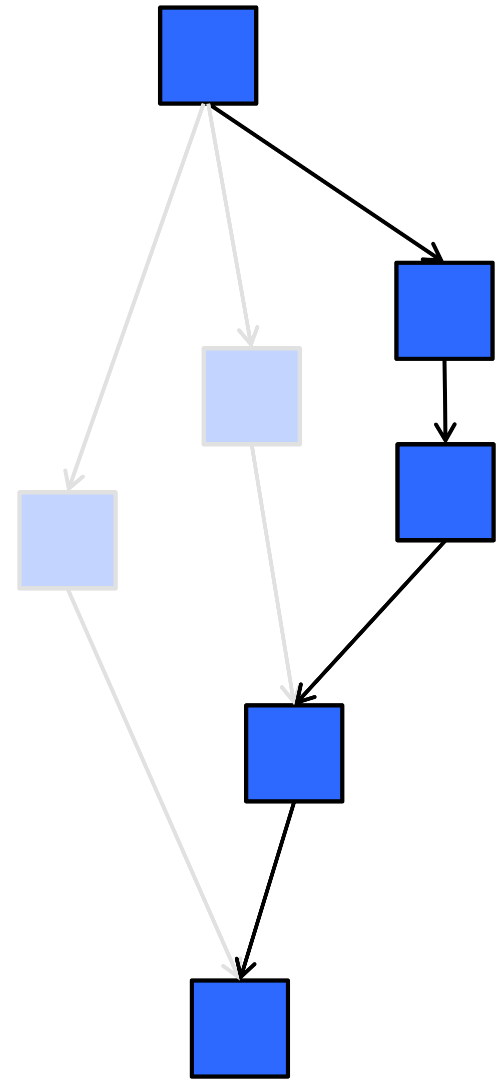
- Includes extra cost for synchronization
- Often dominated by cache line transfers.



Lower Bound on Greedy Scheduling

Work-Span Limit

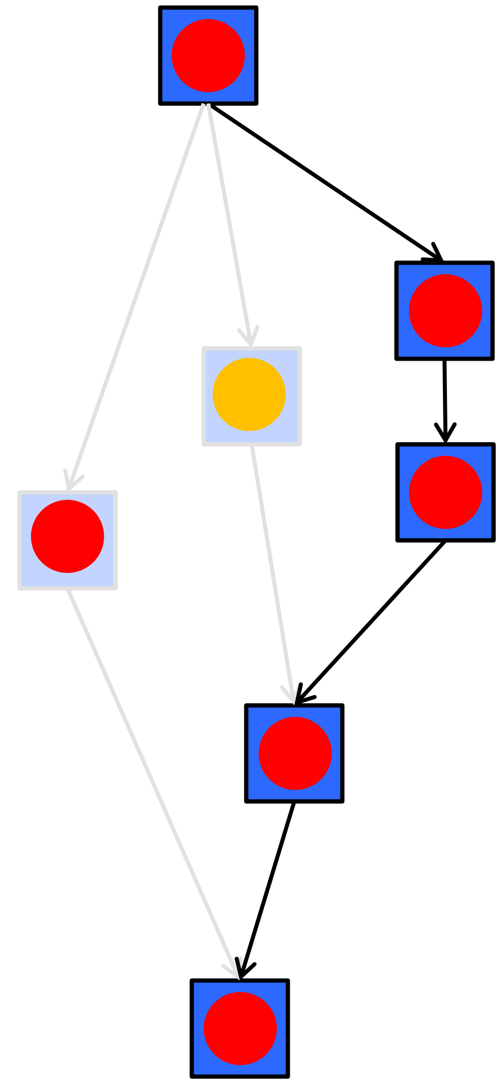
$$\max(T_1/P, T_\infty) \le T_P$$



Upper Bound on Greedy Scheduling

Brent's Lemma

$$T_P \leq (T_1 - T_\infty)/P + T_\infty$$

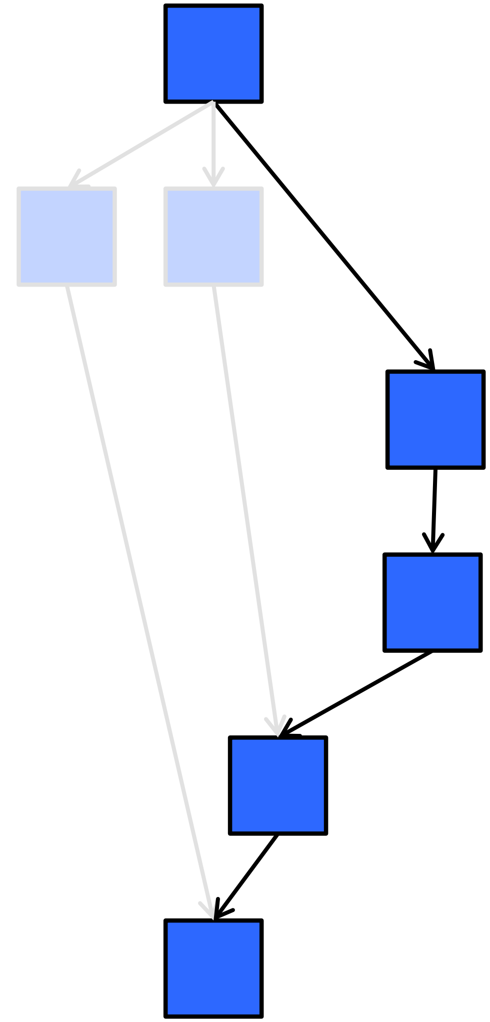


Applying Brent's Lemma to 2 Processors

$T_1 = 7$

$T_\infty = 5$

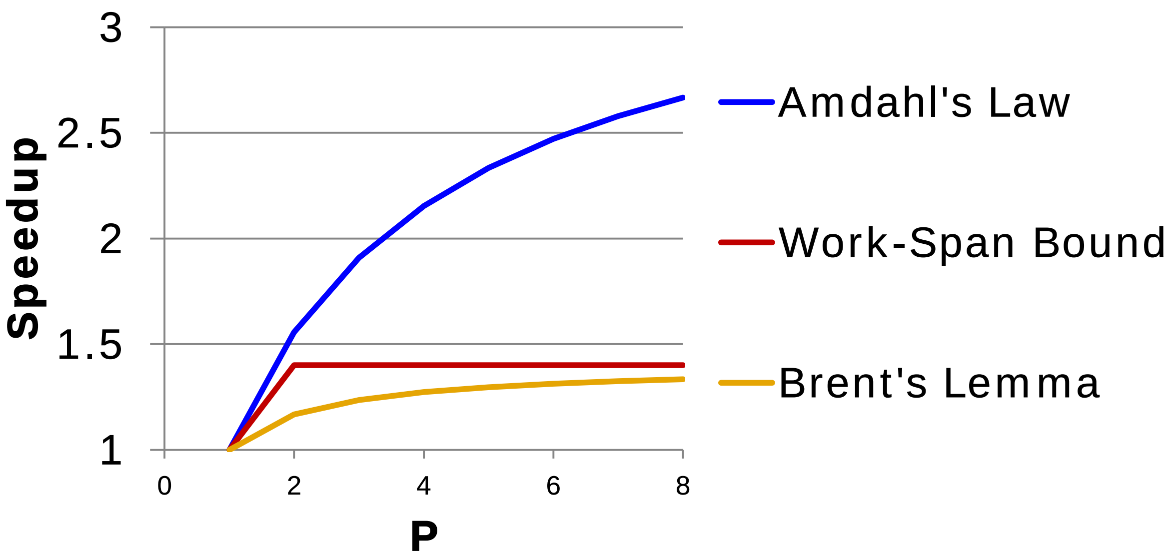
$T_2 \leq (T_1 - T_\infty)/P + T_\infty$
$\leq (7-5)/2 + 5$
≤ 6

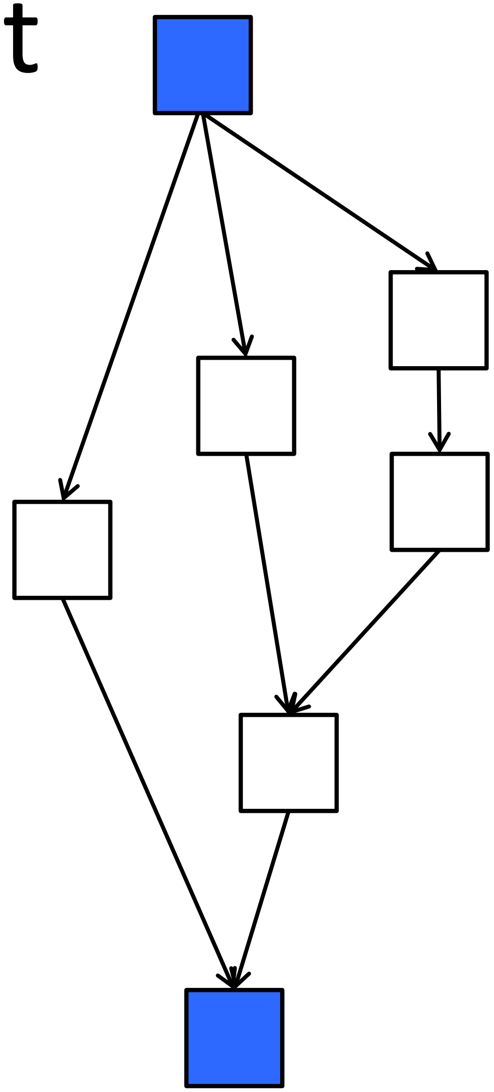


Amdahl Was An Optimist

Amdahl's Law

$$T_{serial} + T_{parallel}/P \leq T_P$$





Estimating Running Time

- Scalability requires that T_∞ be dominated by $T_{1.}$

$$\boxed{T_P \approx T_1/P + T_\infty \quad \text{if} \quad T_\infty \ll T_1}$$

- Increasing work hurts parallel execution proportionately.
- The span impacts scalability, even for finite P.

Parallel Slack

- Sufficient parallelism implies linear speedup.

$$\boxed{\underbrace{T_P \approx T_1/P}_{\text{Linear speedup}} \quad \text{if} \quad \underbrace{T_1/T_\infty \gg P}_{\textit{Parallel slack}}}$$

Definitions for Asymptotic Notation

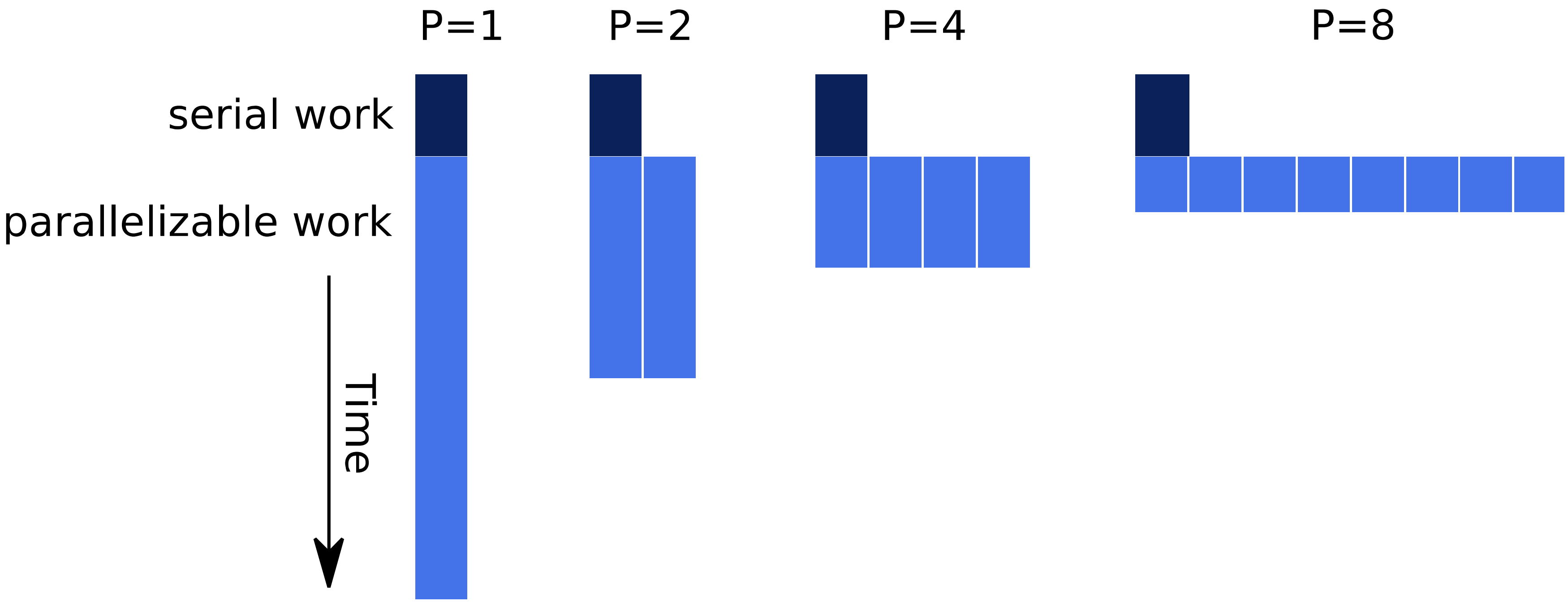
- $T(N) = O(f(N)) \equiv T(N) \leq c \cdot f(N)$ for some constant *c*.
- $T(N) = \Omega(f(N)) \equiv T(N) \geq c \cdot f(N)$ for some constant *c*.
- $T(N) = \Theta(f(N)) \equiv c_1 \cdot f(N) \leq T(N) \leq c_2 \cdot f(N)$ for some constants *c*1 and *c*2.

Quiz: If $T_1(N) = O(N^2)$ and $T_\infty(N) = O(N)$, then T_1/T_∞ = ?

a. O(N)
b. O(1)
c. O(1/N)
d. all of the above
e. need more information

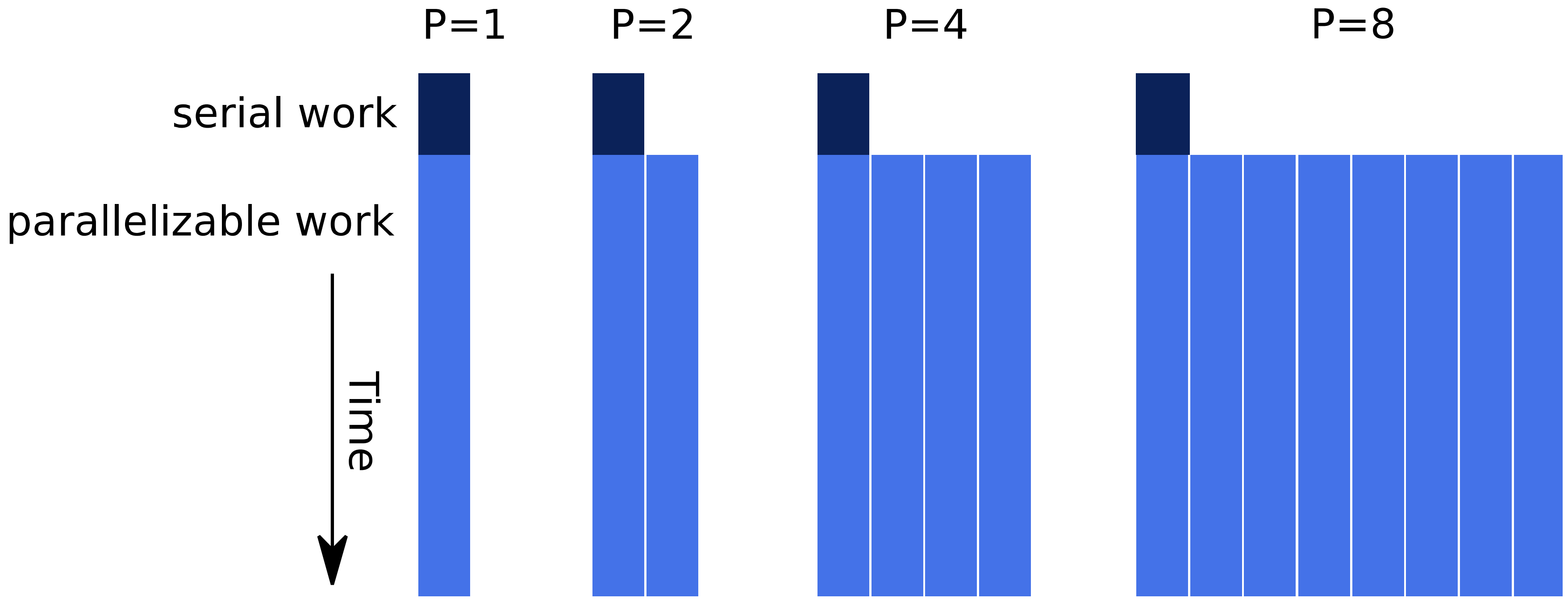
Amdahl vs. Gustafson-Baris

Amdahl



Amdahl vs. Gustafson-Baris

Gustafson-Baris



Optional Versus Mandatory Parallelism

- Task constructs in Intel® TBB and Cilk™ Plus *grant permission* for parallel execution, but do not mandate it.
  - Exception: TBB's std::thread (a.k.a. tbb::tbb_thread)
- Optional parallelism is key to efficiency
  - You provide parallel slack (over decomposition).
  - *Potential parallelism should be greater than physical parallelism.*
  - TBB and Cilk Plus convert potential parallelism to actual parallelism as appropriate.

A task is an *opportunity* for parallelism

Reminder of Some Assumptions

- Memory bandwidth is not a limiting resource.
- There is no speculative work.
- The scheduler is greedy.

INTEL® CILK™ PLUS AND INTEL® THREADING BUILDING BLOCKS (TBB)

Course Outline

- Introduction
  - Motivation, goals, patterns
- Background
  - Machine model, complexity, work-span
- **Cilk™ Plus and Threading Building Blocks**
  - **Programming model**
  - **Examples**
- Practical matters
  - Debugging and profiling

Cilk™ Plus and TBB: Outline

- Feature summaries
- C++ review
- Map pattern
- Reduction pattern
- Fork-join pattern
- Example: polynomial multiplication
- Complexity analysis
- Pipeline pattern

Full code examples for these patterns can be downloaded from http://parallelbook.com/downloads

Summary of Cilk™ Plus

Thread Parallelism
cilk_spawn
cilk_sync
cilk_for

Vector Parallelism
array notation
#pragma simd
elemental functions

Reducers
reducer
reducer_op{add,and,or,xor}
reducer_{min,max}{_index}
reducer_list_{append,prepend}
reducer_ostream
reducer_string
holder

TBB 4.0 Components

Parallel Algorithms
parallel_for
parallel_for_each
parallel_invoke
parallel_do
parallel_scan
parallel_sort
parallel_[deterministic]_reduce

Macro Dataflow
parallel_pipeline
tbb::flow::...

Concurrent Containers
concurrent_hash_map
concurrent_unordered_{map,set}
concurrent_[bounded_]queue
concurrent_priority_queue
concurrent_vector

Task scheduler
task_group, structured_task_group
task
task_scheduler_init
task_scheduler_observer

Thread Local Storage
combinable
enumerable_thread_specific

Synchronization Primitives
atomic, condition_variable
[recursive_]mutex
{spin,queuing,null} [_rw]_mutex
critical_section, reader_writer_lock

Memory Allocation
tbb_allocator
zero_allocator
cache_aligned_allocator
scalable_allocator

Threads
std::thread

C++ Review

"Give me six hours to chop down a tree and I will spend the first four sharpening the axe."

- Abraham Lincoln

C++ Review: Half Open Interval

- STL specifies a sequence as a half-open interval [*first*,*last*)
  - *last*–*first* == size of interval
  - *first*==*last* ⇔ empty interval
- If object x contains a sequence
  - x.begin() points to first element.
  - x.end() points to “one past last” element.

```
void PrintContainerOfTypeX( const X& x ) {
   for( X::iterator i=x.begin(); i!=x.end(); ++i )
      cout << *i << endl;
}
```

C++ Review: Function Template

- Type-parameterized function.
  - Strongly typed.
  - Obeys scope rules.
  - Actual arguments evaluated exactly once.
  - Not redundantly instantiated.

```
template<typename T>
void swap( T& x, T& y ) {
    T z = x;
    x = y;
    y = z;
}
```

[first,last) define half-open interval

```
void reverse( float* first, float* last ) {
    while( first<last-1 )
        swap( *first++, *--last );
}
```

Compiler instantiates template **swap** with T=float.

Genericity of swap

```
template<typename T>
void swap( T& x, T& y ) {
    T z = x;
    x = y;
    y = z;
}
```

Construct a copy

Assign

Destroy z

C++03 Requirements for T

T(const T&)	Copy constructor
void T::operator=(const T&)	Assignment
~T()	Destructor

C++ Review: Template Class

- Type-parameterized class

```
template<typename T, typename U>
class pair {
public:
    T first;
    U second;
    pair( const T& x, const U& y ) : first(x), second(y) {}
};
```

```
pair<string,int> x;
x.first = "abc";
x.second = 42;
```

Compiler instantiates template **`pair`** with T=string and U=int.

C++ Function Object

- Also called a “functor”
- Is object with member operator().

```
class LinearOp {
    float a, b;
public:
    float operator() ( float x ) const {return a*x+b;}
    Linear( float a_, float b_ ) : a(a_), b(b_) {}
};
```

```
LinearOp f(2,5);
y = f(3);
```

Could write as
y = f.operator()(3);

Template Function + Functor = Flow Control

```
template<typename I, typename Func>
void ForEach( I lower, I upper, const Func& f ) {
    for( I i=lower; i<upper; ++i )
        f(i);
}
```

Template function for iteration

```
class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate( float& acc_, float* src_ ) : acc(acc_), src(src_) {}
    void operator()( int i ) const {acc += src[i];}
};
```

Functor

```
float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}
```

Pass functor to template function. Functor becomes “body” of control flow “statement”.

So Far

- Abstract control structure as a template function.
- Encapsulate block of code as a functor.
- Template function and functor can be arbitrarily complex.

Recap: Capturing Local Variables

- Local variables were captured via fields in the functor

Field holds reference to **`sum`**.

Capture reference to **`sum`** in **`acc`**.

```
class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate( float& acc_, float* src_ ) : acc(acc_), src(src_) {}
    void operator()( int i ) const {acc += src[i];}
};
float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}
```

Use reference to **`sum`**.

Formal parameter **`acc_`** bound to local variable **`sum`**

Array Can Be Captured as Pointer Value

Field for capturing **a** declared as a pointer.

```
class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate( float& acc_, float* src_ ) : acc(acc_), src(src_) {}
    void operator()( int i ) const {acc += src[i];}
};
```

```
float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}
```

a implicitly converts to pointer

An Easier Naming Scheme

- Name each field and parameter after the local variable that it captures.

This is tedious mechanical work. Can we make the compiler do it?

```
class Accumulate {
    float& sum;
    float* a;
public:
    Accumulate( float& sum_, float* a_ ) : sum(sum_), a(a_) {}
    void operator()( int i ) const {sum += a[i];}
};
```

```
float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, Accumulate(sum,a) );
    return sum;
}
```

C++11 Lambda Expression

- Part of C++11
- Concise notation for functor-with-capture.
- Available in recent Intel, Microsoft, GNU C++, and clang++ compilers.

Intel Compiler Version

		11.*	12.*	13.*, 14.*
Platform	Linux* OS	`-std=c++0x`		`-std=c++11` (or `–std=c++0x`)
	Mac* OS			
	Windows* OS	`/Qstd:c++0x`	on by default	

With Lambda Expression

```
~~class Accumulate {
    float& acc;
    float* src;
public:
    Accumulate( float& acc_, float* src_ ) : acc(acc_), src(src_) {}
    void operator()( int i ) const {acc += src[i];}
};~~
```

```
float Example() {
    float a[4] = {1,3,9,27};
    float sum = 0;
    ForEach( 0, 4, [&]( int i ) {sum += a[i];} );
    return sum;
}
```

Compiler automatically defines custom *functor* type tailored to capture **`sum`** and **`a`**.

[&] introduces lambda expression that constructs instance of *functor*.

Parameter list and body for *functor*::operator()

Lambda Syntax

[*capture_mode***] (***formal_parameters***) ->** *return_type* **{***body***}**

[&] ⇒ by-reference
[=] ⇒ by-value
[] ⇒ no capture

Can omit if there are no parameters *and* return type is implicit.

Can omit if return type is void or *code* is “return *expr*;”

Examples

```
[&](float x) {sum+=x;}
```

```
[&]{return *p++;}
```

```
[=](float x) {return a*x+b;}
```

```
[]{return rand();}
```

```
[](float x, float y)->float {
   if(x<y) return x;
   else return y;
}
```

Not covered here: how to specify capture mode on a per-variable basis.

Note About Anonymous Types

- Lambda expression returns a functor with *anonymous type*.
  - Thus lambda expression is typically used only as argument to template function or with C++11 auto keyword.
  - Later we'll see two other uses unique to Cilk Plus.

```
template<typename F>
void Eval( const F& f ) {
   f();
}

void Example1() {
   Eval( []{printf("Hello, world\n");} );
}
```

Template deduces functor's type instead of specifying it.

Expression []{...} has anonymous type.

Compiler deduces type of **f** from right side expression.

```
void Example2() {
   auto f = []{printf("Hello, world\n");};
   f();
}
```

Note on Cilk™ Plus Keywords

- Include <cilk/cilk.h> to get nice spellings

In <cilk/cilk.h>

```
#define cilk_spawn _Cilk_spawn
#define cilk_sync _Cilk_sync
#define cilk_for _Cilk_for
```

```
// User code
#include <cilk/cilk.h>
int main() {
    cilk_for( int i=0; i<10; ++i ) {
        cilk_spawn f();
        g();
        cilk_sync;
    }
}
```

Cilk™ Plus Elision Property

- Cilk program has corresponding serialization
  - Equivalent to executing program with single worker.
- Different ways to force serialization:
  - #include **<cilk/cilk_stub.h>** at top of source file

In <cilk/cilk_stub.h>

```
#define _Cilk_sync
#define _Cilk_spawn
#define _Cilk_for for
```

  - Command-line option
    - icc: **-cilk-serialize**
    - icl: **/Qcilk-serialize**
  - Visual Studio:
    - Properties → C/C++ → Language [Intel C++] → Replace Intel Cilk Plus Keywords with Serial Equivalents

Note on TBB Names

- Most public TBB names reside in namespace **tbb**

```
#include "tbb/tbb.h"
using namespace tbb;
```

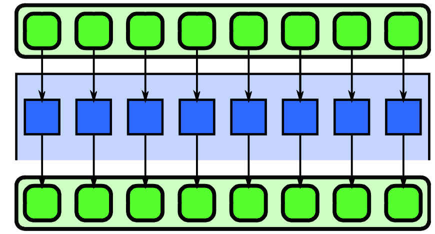
- C++11 names are in namespace **std**.

```
#include "tbb/compat/condition_variable"
#include "tbb/compat/thread"
#include "tbb/compat/tuple"
```

- Microsoft PPL names can be injected from namespace **tbb** into namespace **Concurrency**.

```
#include "tbb/compat/ppl.h"
```

Map Pattern



Intel® Cilk™ Plus

```
a[0:n] = f(b[0:n]);
```

```
#pragma simd
for( int i=0; i<n; ++i )
    a[i] = f(b[i]);
```

```
cilk_for( int i=0; i<n; ++i )
    a[i] = f(b[i]);
```

Intel®TBB

```
parallel_for( 0, n, [&]( int i ) {
    a[i] = f(b[i]);
});
```

```
parallel_for(
    blocked_range<int>(0,n),
    [&](blocked_range<int> r ) {
        for( int i=r.begin(); i!=r.end(); ++i )
            a[i] = f(b[i]);
    });
```

Map in Array Notation

- Lets you specify parallel intent
  - Give license to the compiler to vectorize

```
// Set y[i] ← y[i] + a ·x[i] for i∈[0..n)
void saxpy(float a, float x[], float y[], size_t n ) {
    y[0:n] += a*x[0:n];
}
```

Array Section Notation

Pointer or array

base[*first*:*length*:*stride*]

First index

Number of elements (different than F90!)

Optional stride

Rules for *section*$_1$ *op section*$_2$

- Elementwise application of *op*
- Also works for *func*(*section*$_1$, *section*$_2$)
- Sections must be the same length
- Scalar arguments implicitly extended

More Examples

- Rank 2 Example – Update m×n tile with corner [i][j].

```
Vx[i:m][j:n] += a*(U[i:m][j+1:n]-U[i:m][j:n]);
```

Scalar implicitly extended

- Function call

```
theta[0:n] = atan2(y[0:n],1.0);
```

- Gather/scatter

```
w[0:n] = x[i[0:n]];
y[i[0:n]] = z[0:n];
```

Improvement on Fortran 90

- Compiler does *not* generate temporary arrays.
  - Would cause unpredictable space demands and performance issues.
  - Want *abstraction with minimal penalty*.
  - Partial overlap of left and right sides is undefined.
- Exact overlap still allowed for updates.
  - Just like for structures in C/C++.

```
x[0:n] = 2*x[1:n];          // Undefined – partial overlap*
x[0:n] = 2*x[0:n];          // Okay – exact overlap
x[0:n:2] = 2*x[1:n:2];      // Okay – interleaved
```

*unless n≤1.

Mapping a Statement with Array Notation

```
template<typename T>
T* destructive_move( T* first, T* last, T* output ) {
    size_t n = last-first;
    []( T& in, T& out ){
        out = std::move(in);
        in.~T();
    } ( first[0:n], output[0:n] );
    return output+n;
}
```

#pragma simd

- Another way to specify vectorization
  - Ignorable by compilers that do not understand it.
  - Similar in style to OpenMP "**`#pragma parallel for`**"

```
void saxpy( float a, float x[], float y[], size_t n ) {
#pragma simd
    for( size_t i=0; i<n; ++i )
        y[i] += a*x[i];
}
```

Note: OpenMP 4.0 adopted a similar "#pragma omp simd"

Clauses for Trickier Cases

- **`linear`** clause for induction variables
- **`private`**, **`firstprivate`**, **`lastprivate`** à la OpenMP

```
void zip( float *x, float *y, float *z, size_t n ) {
#pragma simd linear(x,y,z:2)
    for( size_t i=0; i<n; ++i ) {
        *z++ = *x++;
        *z++ = *y++;
    }
}
```

z has step of 2 per iteration.

Elemental Functions

- Enables vectorization of separately compiled scalar callee.

In file with definition.

```
__declspec(vector)
float add(float x, float y) {
    return x + y;
}
```

In file with call site.

```
__declspec(vector) float add(float x, float y);

void saxpy( float a, float x[], float y[], size_t n ) {
#pragma simd
    for( size_t i=0; i<n; ++i )
        y[i] = add(y[i], a*x[i]);
}
```

Final Comment on Array Notation and `#pragma simd`

- No magic – just does tedious bookkeeping.
- Use "structure of array" (SoA) instead of "array of structure" (AoS) to get SIMD benefit.

cilk_for

- A way to specify thread parallelism.

```
void saxpy( float a, float x[], float y[], size_t n ) {
    cilk_for( size_t i=0; i<n; ++i )
        y[i] += a*x[i];
}
```

Syntax for cilk_for

- Has restrictions so that iteration space can be computed before executing loop.

!=
<
>
>=
<=

limit and *stride* might be evaluated only once.

Must be integral type or random access iterator

index relop limit
limit relop index

index += stride
index -= stride
index++
++index
index--
--index
index

cilk_for(*type index* = *expr*; *condition*; *incr*)
    body;

iterations must be okay to execute in parallel.

Controlling grainsize

- By default, cilk_for tiles the iteration space.
  - Thread executes entire tile
  - Avoids excessively fine-grained synchronization
- For severely unbalanced iterations, this might be suboptimal.
  - Use **pragma cilk grainsize** to specify size of a tile

```
#pragma cilk grainsize = 1
cilk_for( int i=0; i<n; ++i )
    a[i] = f(b[i]);
```

tbb::parallel_for

- Has several forms.

Execute *functor*(*i*) for all $i \in$ [*lower*,*upper*)

parallel_for(*lower, upper, functor*);

Execute *functor*(*i*) for all $i \in$ {*lower*,*lower*+*stride*,*lower*+2**stride*,...}

parallel_for(*lower, upper, stride, functor*);

Execute *functor*(*subrange*) for all *subrange* in *range*

parallel_for(*range, functor*);

Range Form

```
template <typename Range, typename Body>
void parallel_for(const Range& r, const Body& b );
```

- Requirements for a Range type R:

R(const R&)	Copy a range
R::~R()	Destroy a range
bool R::empty() const	Is range empty?
bool R::is_divisible() const	Can range be split?
R::R (R& r, split)	Split r into two subranges

- Enables parallel loop over any recursively divisible range. Library provides **blocked_range**, **blocked_range2d**, **blocked_range3d**
- Programmer can define new kinds of ranges
- Does not have to be dimensional!

2D Example

```
// serial
for( int i=0; i<m; ++i )
    for( int j=0; j<n; ++j )
        a[i][j] = f(b[i][j]);
```

Does 2D tiling, hence better cache usage in some cases than nesting 1D parallel_for.

```
parallel_for(
    blocked_range2d<int>(0,m,0,n),
    [&](blocked_range2d<int> r ) {
        for( int i=r.rows ().begin(); i!=r.rows().end(); ++i )
            for( int j=r.rows().begin(); j!=r.cols().end(); ++j )
                a[i][j] = f(b[i][j]);
    });
```

Optional *partitioner* Argument

Recurse all the way down *range.*

```
tbb::parallel_for( range, functor, tbb::simple_partitioner() );
```

Choose recursion depth heuristically.

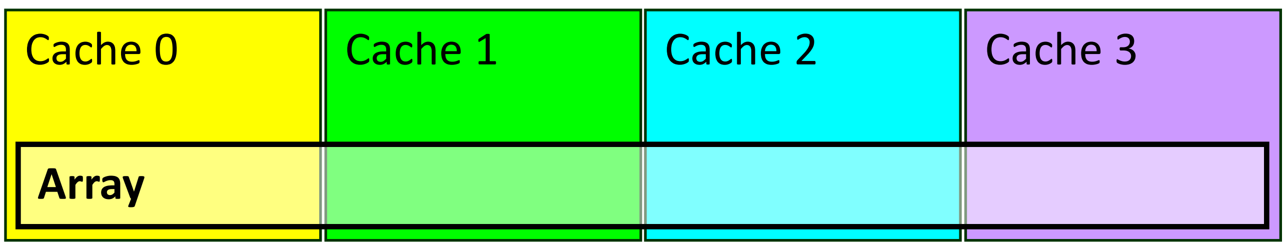
```
tbb::parallel_for( range, functor, tbb::auto_partitioner() );
```

Replay with cache optimization.

```
tbb::parallel_for( range, functor, affinity_partitioner );
```

Iteration↔Thread Affinity

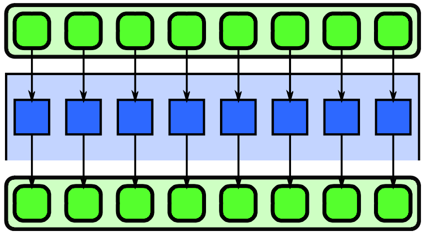
- Big win for serial repetition of a parallel loop.
  - Numerical relaxation methods
  - Time-stepping marches



(Simple model of separate cache per thread)

```
affinity_partitioner ap;
...
for( t=0; ...; t++ )
    parallel_for(range, body, ap);
```

Map Recap



Intel® Cilk™ Plus

```
cilk_for( int i=0; i<n; ++i )
    a[i] = f(b[i]);
```

Thread parallelism

```
a[0:n] = f(b[i:n]);
```

```
#pragma simd
for( int i=0; i<n; ++i )
    a[i] = f(b[i]);
```

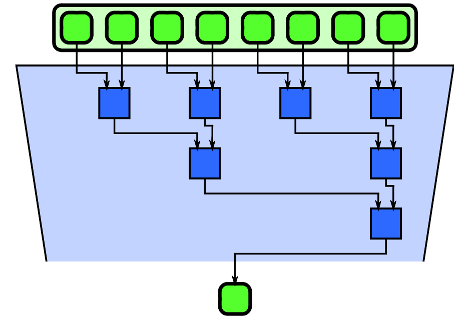
Vector parallelism

Intel®TBB

```
parallel_for( 0, n, [&]( int i ) {
    a[i] = f(b[i]);
});
```

```
parallel_for(
    blocked_range<int>(0,n),
    [&](blocked_range<int> r ) {
        for( int i=r.begin(); i!=r.end(); ++i )
            a[i] = f(b[i]);
    });
```

Reduction Pattern



Intel® Cilk™ Plus

```
float sum = __sec_reduce_add(a[i:n]);
```

```
#pragma simd reduction(+:sum)
float sum=0;
for( int i=0; i<n; ++i )
    sum += a[i];
```

```
cilk::reducer<op_add<float> > sum = 0;
cilk_for( int i=0; i<n; ++i )
    *sum += a[i];
... = sum.get_value();
```

Intel®TBB

```
enumerable_thread_specific<float> sum;
parallel_for( 0, n, [&]( int i ) {
    sum.local() += a[i];
});
... = sum.combine(std::plus<float>());
```

```
sum = parallel_reduce(
    blocked_range<int>(0,n),
    0.f,
    [&](blocked_range<int> r, float s) -> float
    {
        for( int i=r.begin(); i!=r.end(); ++i )
            s +=  a[i];
        return s;
    },
    std::plus<float>()
);
```

Reduction in Array Notation

- Build-in reduction operation for common cases

  +, *, min, index of min, etc.
- User-defined reductions allowed too.

```
float dot( float x[], float y[], size_t n ) {
    return __sec_reduce_add( x[0:n]*y[0:n] );
}
```

sum reduction

elementwise multiplication

Reduction with `#pragma simd`

- **`reduction`** clause for reduction variable

```
float dot( float x[], float y[], size_t n ) {
    float sum = 0;
#pragma simd reduction(+:sum)
    for( size_t i=0; i<n; ++i )
        sum += x[i]*y[i];
    return sum;
}
```

Indicates that loop performs + reduction with **`sum`**.

Reducers in Cilk Plus

- Enable lock-free use of shared reduction locations
  - Work for any associative reduction operation.
  - Reducer does *not* have to be local variable.

Slides use new icc 14.0 syntax for reducers.

Not lexically bound to a particular loop.

```
cilk::reducer_opadd<float> sum = 0;
...
cilk_for( size_t i=1; i<n; ++i )
    *sum += f(i);
... = sum.get_value();
```

Updates local view of **sum**.

Get global view of **sum**.

Reduction with `enumerable_thread_specific`

- Good when:
  - Operation is commutative
  - Operand type is big

Container of thread-local elements.

```
enumerable_thread_specific<BigMatrix> sum;
...
parallel_for( 0, n, [&]( int i ) {
    sum.local() += a[i];
});

... = sum.combine(std::plus<BigMatrix>());
```

Get thread-local element of **sum**.

Return reduction over thread-local elements.

Reduction with `parallel_reduce`

- Good when
  - Operation is non-commutative
  - Tiling is important for performance

```
sum = parallel_reduce(
    blocked_range<int>(0,n),
    0.f,
    [&](blocked_range<int> r, float s) -> float
    {
        for( int i=r.begin(); i!=r.end(); ++i )
            s +=  a[i];
        return s;
    },
    std::plus<float>()
);
```

Recursive range

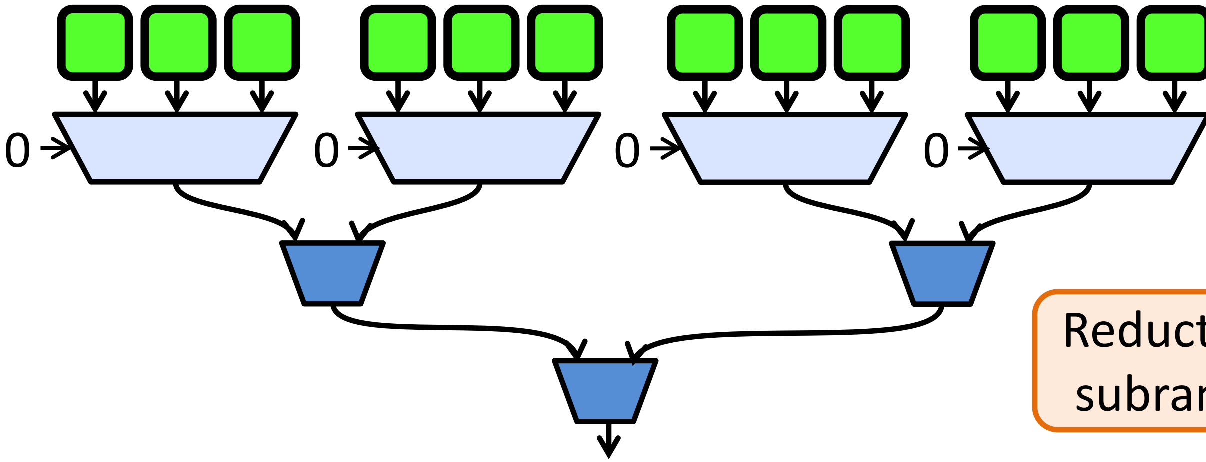
Identity value.

Reduce subrange

Initial value for reducing subrange r. Must be included!

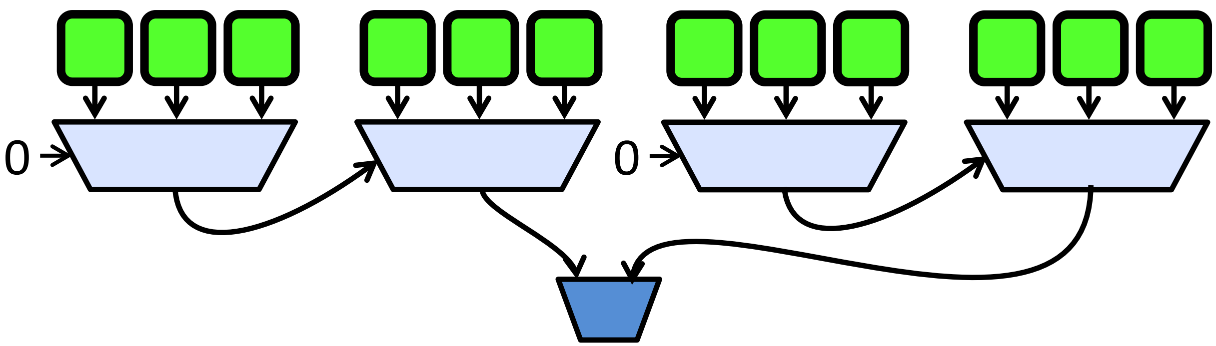
Functor for combining subrange results.

How `parallel_reduce` works



Reduce subranges

Reduction tree for subrange results.



Chaining of subrange reductions.

Notes on `parallel_reduce`

- Optional *partitioner* argument can be used, same as for **parallel_for**
- There is another tiled form that avoids almost all copying overhead.
  - C++11 “move semantics” offer another way to avoid the overhead.
- **parallel_deterministic_reduce** always does tree reduction.
  - Generates deterministic reduction even for floating-point.
  - Requires specification of grainsize.
  - No partitioner allowed.

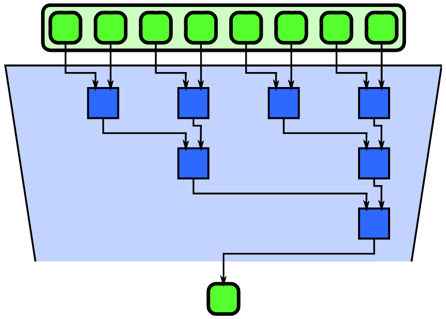
Using parallel_deterministic_reduce

Changed name

```
sum =  parallel_deterministic_reduce (
   blocked_range<int>(0,n,10000),
   0.f,
   [&](blocked_range<int> r, T s) -> float
   {
      for( int i=r.begin(); i!=r.end(); ++i )
         s +=  a[i];
      return s;
   },
   std::plus<T>()
);
```

Added grainsize parameter.
(Default is 1)

Reduction Pattern



Intel® Cilk™ Plus

Intel®TBB

```
cilk::reducer<op_add<float> > sum = 0;
cilk_for( int i=0; i<n; ++i )
   *sum += a[i];
... = sum.get_value();
```

Thread parallelism

```
enumerable_thread_specific<float> sum;
parallel_for( 0, n, [&]( int i ) {
   sum.local() += a[i];
});
... = sum.combine(std::plus<float>());
```

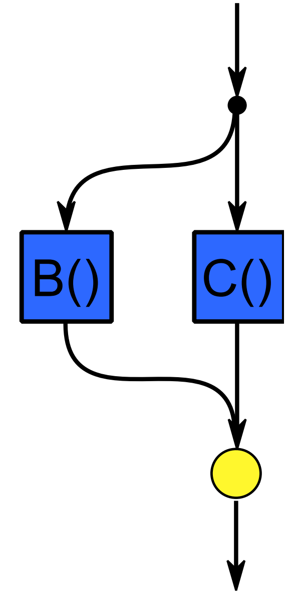
```
sum = parallel_reduce(
   blocked_range<int>(0,n),
   0.f,
   [&](blocked_range<int> r, float s) -> float
   {
      for( int i=r.begin(); i!=r.end(); ++i )
         s +=  a[i];
      return s;
   },
   std::plus<float>()
);
```

```
float sum =
   __sec_reduce_add(a[i:n]);
```

```
#pragma simd reduction(+:sum)
float sum=0;
for( int i=0; i<n; ++i )
   sum += a[i];
```

Vector parallelism

Fork-Join Pattern



Intel® Cilk™ Plus

```
cilk_spawn a();
cilk_spawn b();
c();
cilk_sync();
```

Intel®TBB

```
parallel_invoke( a, b, c );
```

```
task_group g;
g.run( a );
g.run( b );
g.run_and_wait( c );
```

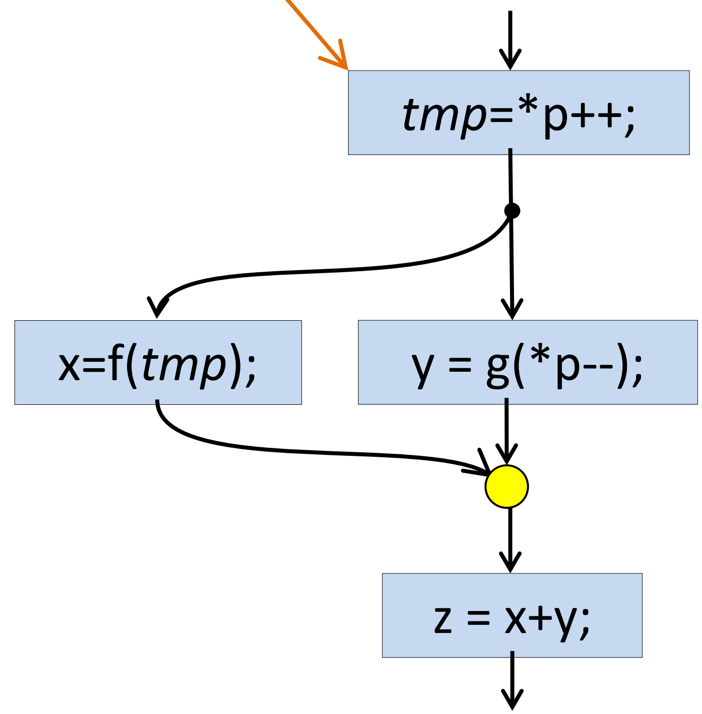
Fork-Join in Cilk Plus

- spawn = asynchronous function call

Optional assignment

```
x = cilk_spawn f(*p++);
y = g(*p--);
cilk_sync;
z = x+y;
```

Arguments to spawned function evaluated *before* fork.

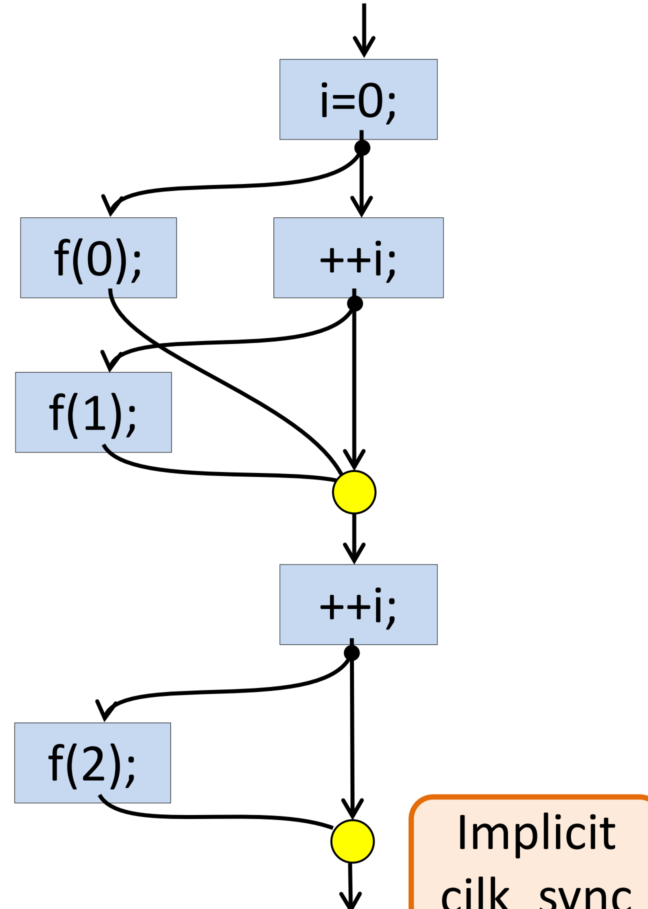


cilk_sync Has Function Scope

- Scope of **cilk_sync** is entire function.
- There is implicit **cilk_sync** at end of a function.

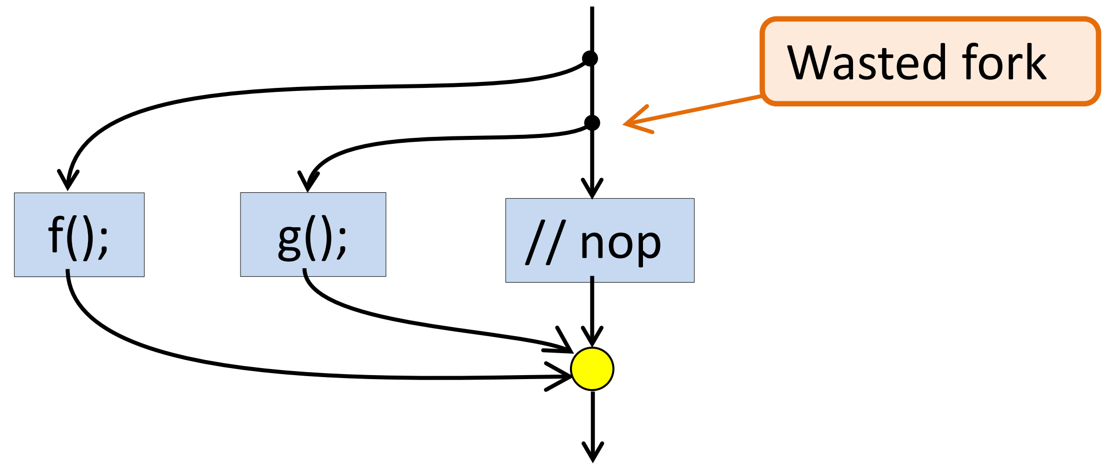
```
void bar() {
    for( int i=0; i<3; ++i ) {
        cilk_spawn f(i);
        if( i&1 ) cilk_sync;
    }
    // implicit cilk_sync
}
```

Serial call/return property:
All Cilk Plus parallelism created by a function completes before it returns.

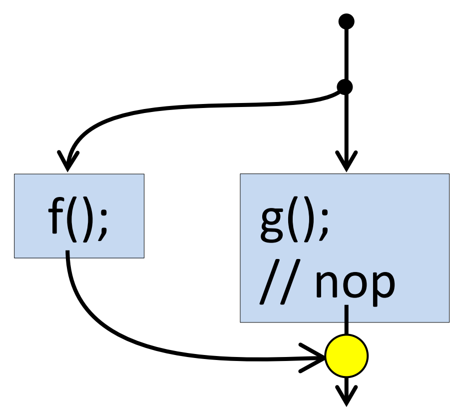


Style Issue

```
// Bad Style
cilk_spawn f();
cilk_spawn g();
// nop
cilk_sync;
```




```
// Preferred style
cilk_spawn f();
g();
// nop
cilk_sync;
```



Spawning a Statement in Cilk Plus

- Just spawn a lambda expression.

```
cilk_spawn [&]{
    for( int i=0; i<n; ++i )
        a[i] = 0;
} ();
...
cilk_sync;
```

Do not forget the ().

Fork-Join in TBB

Useful for *n*-way fork when *n* is small constant.

parallel_invoke($functor_1$, $functor_2$, ...);

task_group g;
...
g.**run**($functor_1$);
...
g.**run**($functor_2$);
...
g.**wait**();

Useful for *n*-way fork when *n* is large or run-time value.

Fine Point About Cancellation/Exceptions

task_group g;
g.**run**(*functor*$_1$);
g.**run**(*functor*$_2$);
functor$_3$();
g.**wait**();

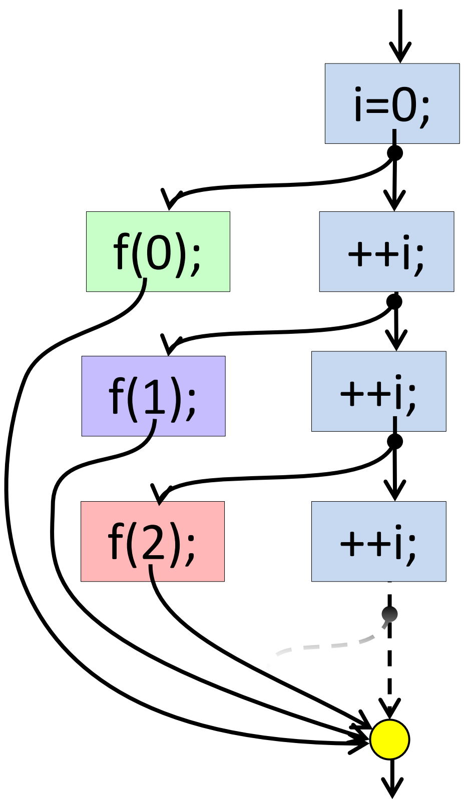
Even if g.**cancel**() is called, *functor*$_3$ still always runs.

task_group g;
g.**run**(*functor*$_1$);
g.**run**(*functor*$_2$);
g.**run_and_wait**(*functor*$_3$);

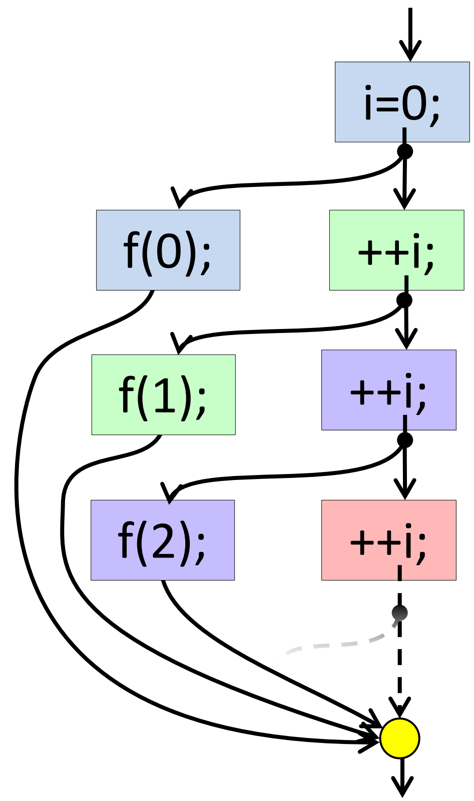
Optimized **run** + **wait**.

Steal Child or Continuation?

```
task_group g;
for( int i=0; i<n; ++i )
    g.run( f(i) );
g.wait();
```

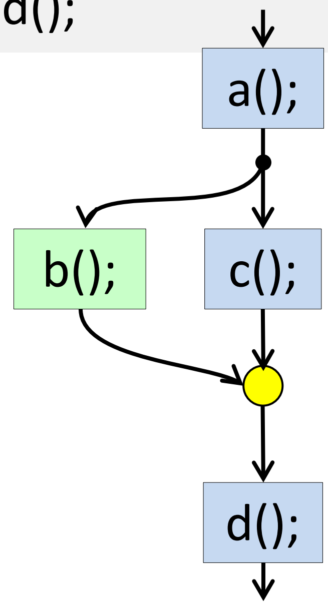



```
for( int i=0; i<n; ++i )
    cilk_spawn f(i);
cilk_sync;
```

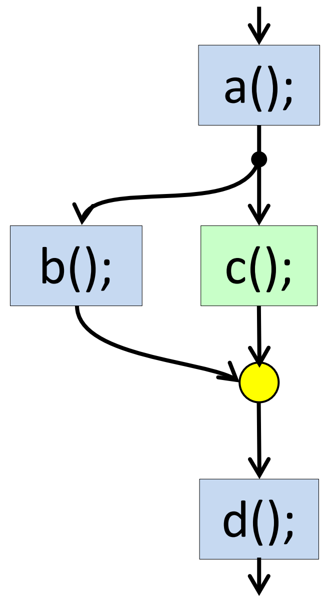


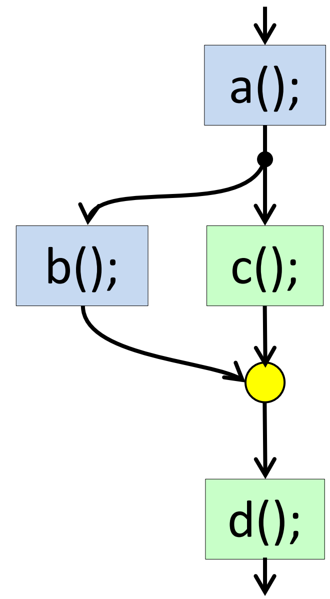
What Thread Continues After a Wait?

```
a();
task_group g;
g.run( b() );
c();
g. wait();
d();
```




```
a();
cilk_spawn b();
c();
cilk_sync;
d();
```



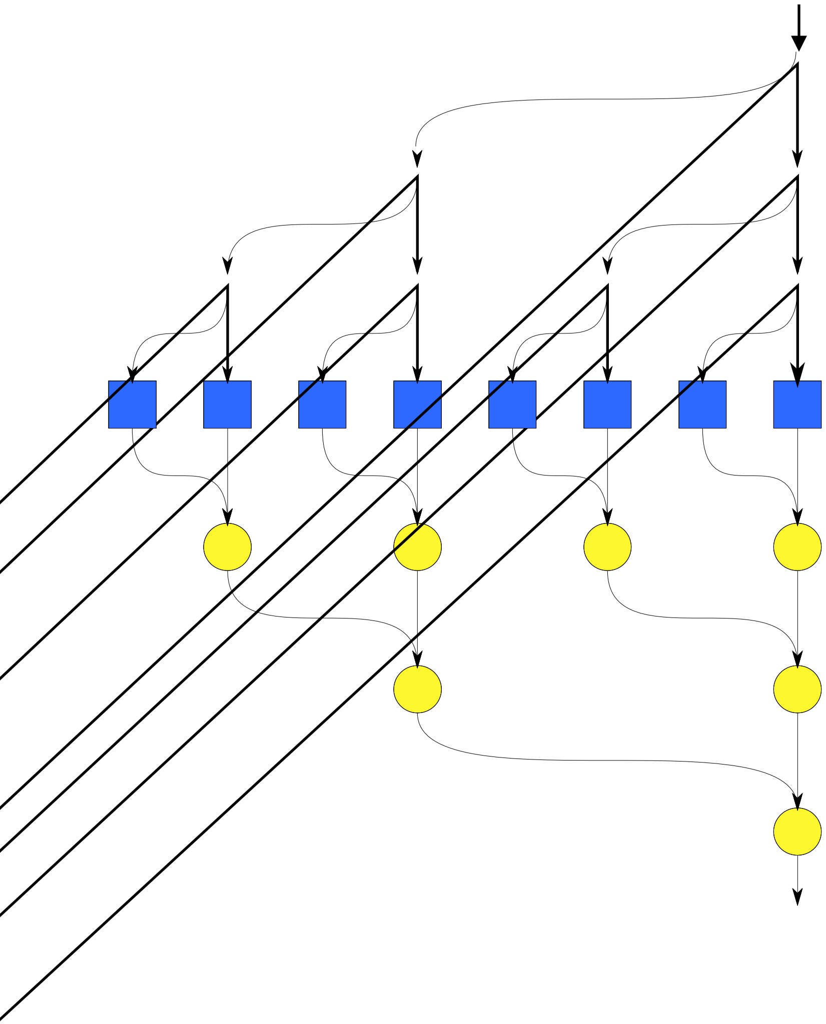


Whichever thread arrives *last* continues execution.

Implications of Mechanisms for Stealing/Waiting

- Overhead
  - Cilk Plus makes unstolen tasks cheap.
  - But Cilk Plus requires compiler support.
- Space
  - Cilk Plus has strong space guarantee: $S_P \leq P \cdot S_1 + P \cdot K$
  - TBB relies on heuristic.
- Time
  - Cilk Plus uses greedy scheduling.
  - TBB is only approximately greedy.
- Thread local storage
  - Function containing cilk_spawn can return on a *different* thread than it was called on.
  - Use reducers in Cilk Plus, not thread local storage.
    - However, Cilk Plus run-time does guarantee that “top level” call returns on same thread.

Fork-Join: Nesting



- Fork-join can be nested
- Spreads cost of work distribution and synchronization.
- This is how **cilk_for** and **tbb::parallel_for** are implemented.

Recursive fork-join enables high parallelism.

Faking Fork-Join in Vector Parallelism

- Using array section to control "if":

```
if( a[0:n] < b[0:n] )
    c[0:n] += 1;
else
    c[0:n] -= 1;
```

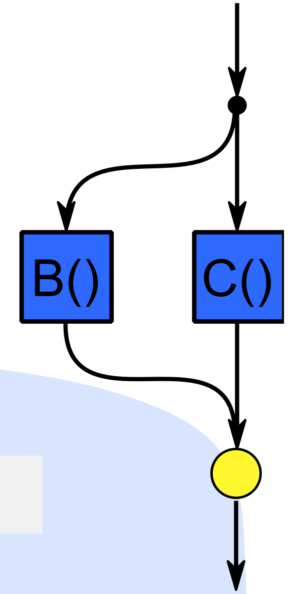
Can be implemented by executing *both* arms of if-else with masking.

- Using **#pragma simd** on loop with "if":

```
#pragma simd
for( int i=0; i<n; ++i )
    if( a[i]<b[i] )
        c[i] += 1;
    else
        c[i] -= 1;
```

Each fork dilutes gains from vectorization.

Fork-Join Pattern



Intel® Cilk™ Plus

Intel®TBB

```
cilk_spawn a();
cilk_spawn b();
c();
cilk_sync();
```

```
parallel_invoke( a, b, c );
```

Thread parallelism

```
task_group g;
g.run( a );
g.run( b );
g.run( c );
g.wait();
```

```
if( x[0:n] < 0 )
    x[0:n] = -x[0:n];
```

```
#pragma simd
for( int i=0; i<n; ++i )
   if( x[i] < 0 )
      x[i] = -x[i];
```

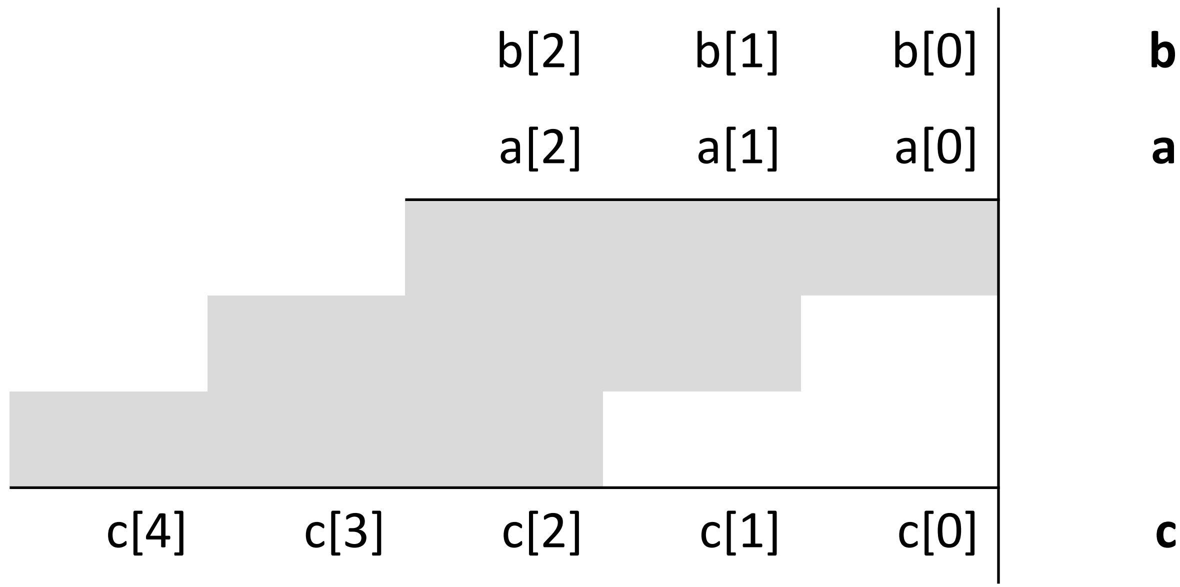
Fake Fork-Join for Vector parallelism

Polynomial Multiplication

Example: c = a·b

$$
\begin{array}{ccccccccc|c}
 & & & & x^2 & + & 2x & + & 3 & \mathbf{b} \\
 & & & & x^2 & + & 4x & + & 5 & \mathbf{a} \\
\cline{5-9}
 & & & & 5x^2 & + & 10x & + & 15 & \\
 & & 4x^3 & + & 8x^2 & + & 12x & & & \\
x^4 & + & 2x^3 & + & 3x^2 & & & & & \\
\hline
x^4 & + & 6x^3 & + & 16x^2 & + & 22x & + & 15 & \mathbf{c}
\end{array}
$$

Storage Scheme for Coefficients



Vector Parallelism with Cilk Plus Array Notation

vector initialization

```
void simple_mul( T c[], const T a[], const T b[], size_t n ) {
    c[0:2*n-1] = 0;
    for (size_t i=0; i<n; ++i)
        c[i:n] += a[i]*b[0:n];
}
```

vector addition

- More concise than serial version
- Highly parallel: $T_1/T_\infty = n^2/n = \Theta(n)$
- What's not to like about it?

Too Much Work!

	T_1
Grade school method	$\Theta(n^2)$
Karatsuba	$\Theta(n^{1.5})$
FFT method	$\Theta(n \lg n)$

However, the FFT approach has high constant factor.
For n about 32-1024, Karatsuba is a good choice.

Karatsuba Trick: Divide and Conquer

- Suppose polynomials a and b have degree n
  - let $K=x^{\lfloor n/2 \rfloor}$

    $a = a_1 K + a_0$

    $b = b_1 K + b_0$

- Compute:

    $t_0 = a_0 \cdot b_0$

    $t_1 = (a_0 + a_1) \cdot (b_0 + b_1)$

    $t_2 = a_1 \cdot b_1$

- Then

    $a \cdot b \equiv t_2 K^2 + (t_1 - t_0 - t_2) K + t_0$

Partition coefficients.

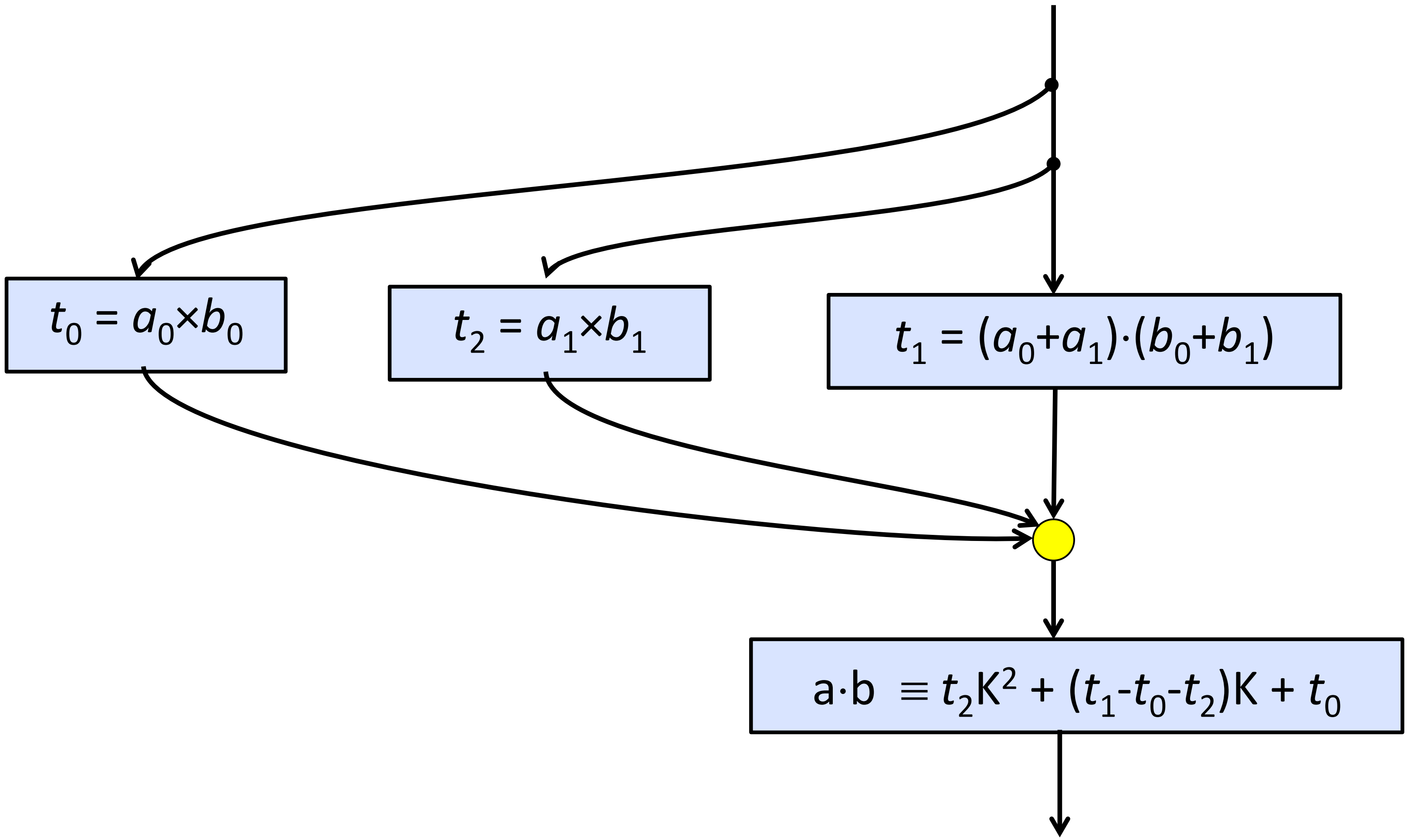
3 half-sized multiplications.
Do these recursively.

Sum products, shifted by multiples of K.

Vector Karatsuba

```
void karatsuba( T c[], const T a[], const T b[], size_t n ) {
    if( n<=CutOff ) {
        simple_mul( c, a, b, n );
    } else {
        size_t m = n/2;
        karatsuba( c, a, b, m );                          // t0 = a0×b0
        karatsuba( c+2*m, a+m, b+m, n-m );                // t2 = a1×b1
        temp_space<T> s(4*(n-m));
        T *a_=s.data(), *b_=a_+(n-m), *t=b_+(n-m);
        a_[0:m] = a[0:m]+a[m:m];                          // a_ = (a0+a1)
        b_[0:m] = b[0:m]+b[m:m];                          // b_ = (b0+b1)
        karatsuba( t, a_, b_, n-m );                      // t1 = (a0+a1)×(b0+b1)

        t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1];          // t = t1-t0-t2
        c[2*m-1] = 0;
        c[m:2*m-1] += t[0:2*m-1];                         // c = t2K^2+(t1-t0-t2)K+t0
    }
}
```

Sub-Products Can Be Computed in Parallel



Multithreaded Karatsuba in Cilk Plus

```
void karatsuba( T c[], const T a[], const T b[], size_t n ) {
    if( n<=CutOff ) {
        simple_mul( c, a, b, n );
    } else {
        size_t m = n/2;
        cilk_spawn karatsuba( c, a, b, m );                        // $t_0 = a_0 \times b_0$
        cilk_spawn karatsuba( c+2*m, a+m, b+m, n-m );              // $t_2 = a_1 \times b_1$
        temp_space<T> s(4*(n-m));
        T *a_=s.data(), *b_=a_+(n-m), *t=b_+(n-m);
        a_[0:m] = a[0:m]+a[m:m];                                   // $a\_ = (a_0+a_1)$
        b_[0:m] = b[0:m]+b[m:m];                                   // $b\_ = (b_0+b_1)$
        karatsuba( t, a_, b_, n-m );                               // $t_1 = (a_0+a_1) \times (b_0+b_1)$
        cilk_sync;
        t[0:2*m-1] -= c[0:2*m-1] + c[2*m:2*m-1];                   // $t = t_1-t_0-t_2$
        c[2*m-1] = 0;
        c[m:2*m-1] += t[0:2*m-1];                                  // $c = t_2K^2+(t_1-t_0-t_2)K+t_0$
    }
}
```

Only change is insertion of Cilk Plus keywords.

Multithreaded Karatsuba in TBB (1/2)

```
void karatsuba(T c[], const T a[], const T b[], size_t n ) {
    if( n<=CutOff ) {
        simple_mul( c, a, b, n );
    } else {
        size_t m = n/2;
        temp_space<T> s(4*(n-m));
        T* t = s.data();
        tbb::parallel_invoke(
            ..., // t0 = a0×b0
            ..., // t2 = a1×b1
            ...  // t1 = (a0+a1)×(b0+b1)
        );
        // t = t1-t0-t2
        for( size_t j=0; j<2*m-1; ++j )
            t[j] -= c[j] + c[2*m+j];
        // c = t2K2+(t1-t0-t2)K+t0
        c[2*m-1] = 0;
        for( size_t j=0; j<2*m-1; ++j )
            c[m+j] += t[j];
    }
}
```

Declared **`temp_space`** where it can be used after **parallel_invoke.**

Three-way fork-join specified with **parallel_invoke**

Explicit loops replace Array Notation.

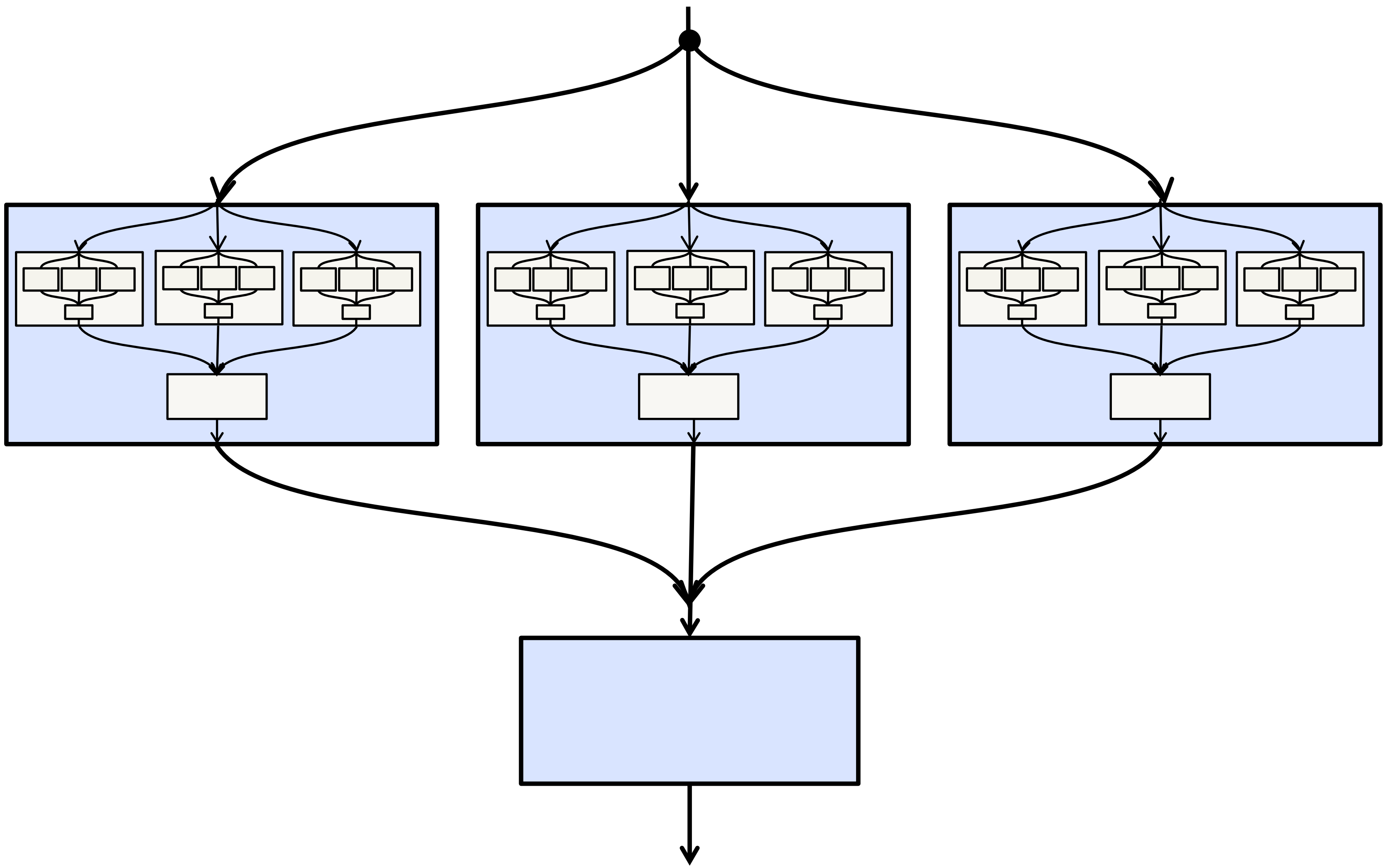
Multithreaded Karatsuba in TBB (2/2)

```
void karatsuba(T c[], const T a[], const T b[], size_t n ) {
      ...
      tbb::parallel_invoke(
         [&] {
            karatsuba( c, a, b, m );
         },
         [&] {
            karatsuba( c+2*m, a+m, b+m, n-m );          // t_2 = a_1×b_1
         },
         [&] {
            T *a_=t+2*(n-m), *b_=a_+(n-m);
            for( size_t j=0; j<m; ++j ) {
               a_[j] = a[j]+a[m+j];                     // a_ = (a_0+a_1)
               b_[j] = b[j]+b[m+j];                     // b_ = (b_0+b_1)
            }
            karatsuba( t, a_, b_, n-m );                // t_1 = (a_0+a_1)×(b_0+b_1)
         }
      );
      ...
}
```

“Capture by reference” here because arrays are being captured.

Another explicit loop.

Parallelism is Recursive

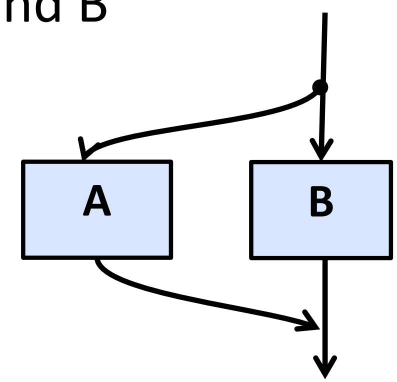


Work-Span Analysis for Fork-Join

- Let B||C denote the fork-join composition of A and B

$$T_1(A||B) = T_1(A) + T_1(B)$$

$$T_\infty(A||B) = \max(T_\infty(A), T_\infty(B))$$



Master Method

If equations have this form:

$T(N) = aT(N/b) + cN^d$

$T(1) = e$

Then solution is:

$T(N) = \Theta(N^{\log_b a})$ if $\log_b a > d$

$T(N) = \Theta(N^d \lg N)$ if $\log_b a = d$

$T(N) = \Theta(N^d)$ if $\log_b a < d$

Work-Span Analysis for Karatsuba Routine

- Let N be number of coefficients

Equations for work

$T_1(N) = 3T_1(N/2) + cN$

$T_1(1) = \Theta(1)$

Solutions

$T_1(N) = \Theta(N^{\log_2 3})$

Equations for span

$T_\infty(N) = T_\infty(N/2) + cN$

$T_\infty(1) = \Theta(1)$

$T_\infty(N) = \Theta(N)$

Equations almost identical, except for one coefficient.

speedup $= T_1(N)/\ T_\infty(N)$
$= \Theta(N^{0.58\ldots})$

Space Analysis for Karatsuba Routine

- Let N be number of coefficients

Equations for serial space

$S_1(N) = S_1(N/2) + cN$

$S_1(1) = \Theta(1)$

Solutions

$S_1(N) = \Theta(N)$

Equations for parallel space?

$S_\infty(N) \leq 3S_\infty(N/2) + cN$

$S_\infty(1) = \Theta(1)$

$S_\infty(N) = O(N^{\log_2 3})$

But what about the space S_P?

Cilk Plus Bound for Karatsuba Routine

- Cilk Plus guarantees $S_P \leq P{\cdot}S_1 + P{\cdot}K$

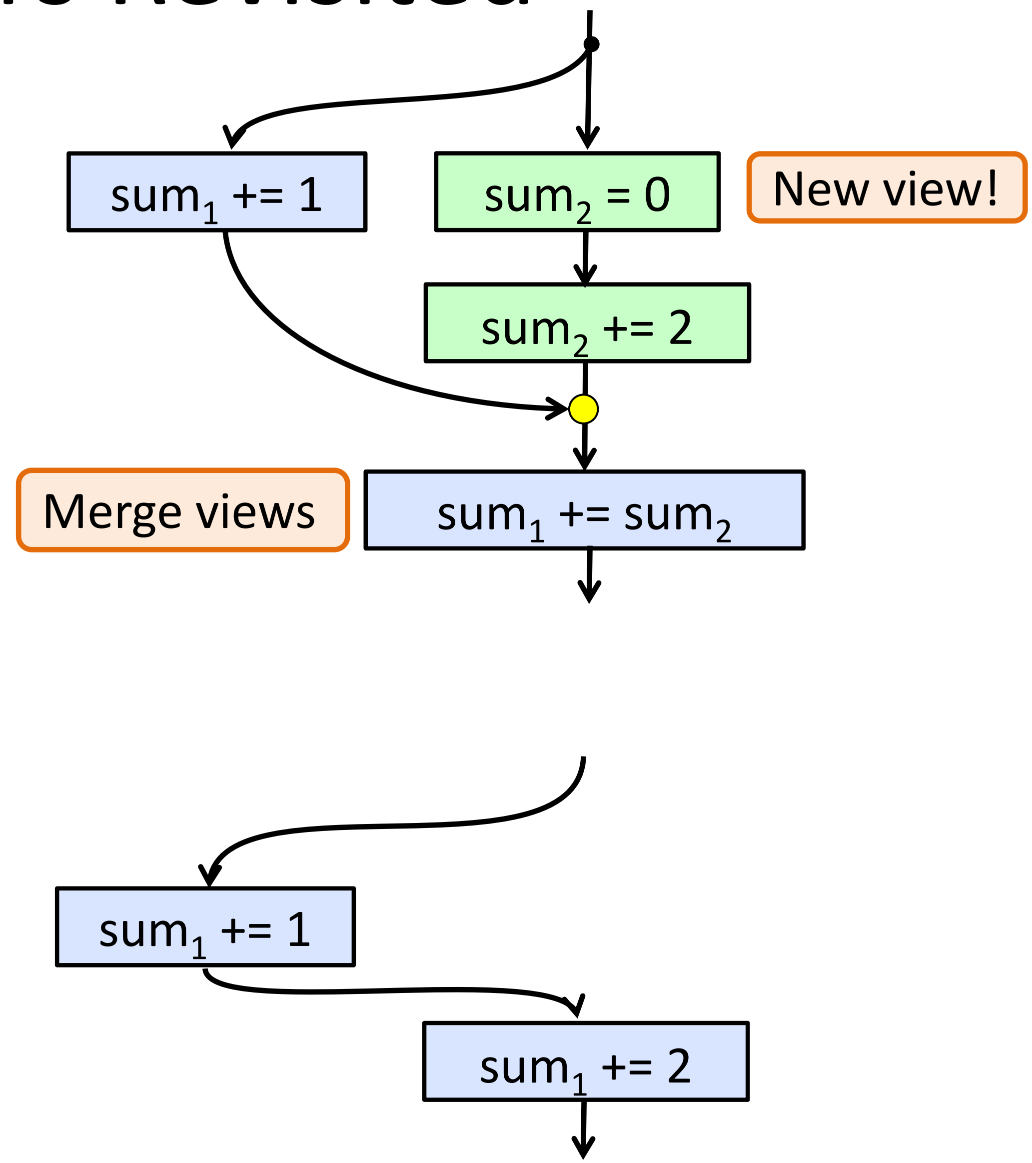
$$\begin{aligned} S_P(N) &= P{\cdot}O(N) + P{\cdot}K \\ &= O(P{\cdot}N) \end{aligned}$$

For small P, a big improvement on the bound $S_\infty(N) = \Theta(N^{\log_2 3})$

Reducers Revisited

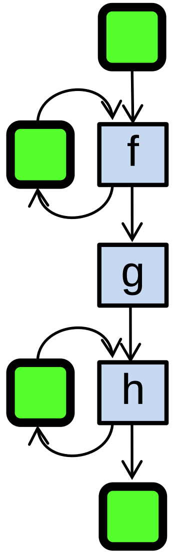
Global variable

```
cilk::reducer_opadd<float> sum;

void f( int m ) {
    *sum += m;
}

float g() {
    cilk_spawn f(1);
    f(2);
    cilk_sync;
    return sum.get_value();
}
```



Reducers enable safe use of global variables *without locks.*

Pipeline Pattern



Intel® Cilk™ Plus (special case)

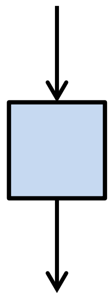
```
S s;
reducer_consume<S,U> sink (
    &s, h
);
...
void Stage2( T x ) {
    sink.consume(g(x));
}
...
while( T x = f() )
    cilk_spawn Stage2(x);
cilk_sync;
```

Intel® TBB

```
parallel_pipeline (
    ntoken,
    make_filter<void,T>(
        filter::serial_in_order,
        [&]( flow_control & fc ) -> T{
            T item = f();
            if( !item ) fc.stop();
            return item;
        }
    ) &
    make_filter<T,U>(
        filter::parallel,
        g
    ) &
    make_filter<U,void>(
        filter:: serial_in_order,
        h
    )
);
```

Serial vs. Parallel Stages

Parallel stage is functional transform.

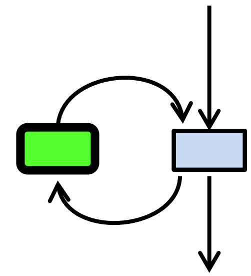


<from,to>

```
make_filter<X,Y>(
    filter::parallel,
    []( X x ) -> Y {
        Y y = foo(x);
        return y;
    }
)
```

You must ensure that parallel invocations of `foo` are safe.

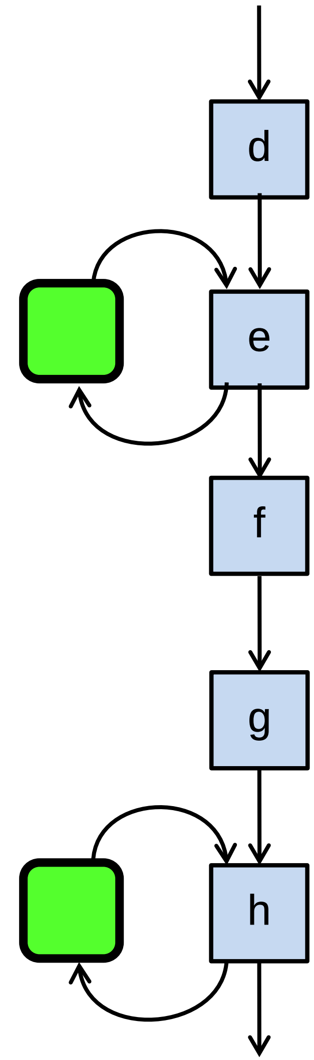
Serial stage has associated state.



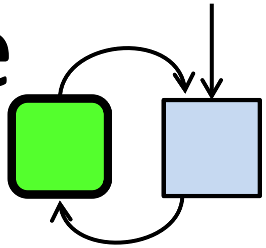
```
make_filter<X,Y>(
    filter::serial_in_order,
    [&]( X x ) -> Y {
        extern int count;
        ++count;
        Y y = bar(x);
        return y;
    }
)
```

In-Order vs. Out-of-Order Serial Stages

- Each in-order stage receives values in the order the previous in-order stage returns them.



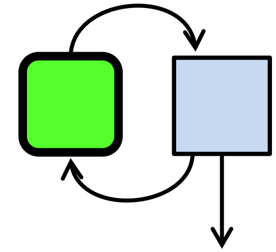
Special Rule for Last Stage



“to” type is **void**

```
make_filter<X,void>(
   filter::serial_in_order,
   [&]( X x ) {
      cout << x;
   }
)
```

Special Rules for First Stage



“from” type is **void**

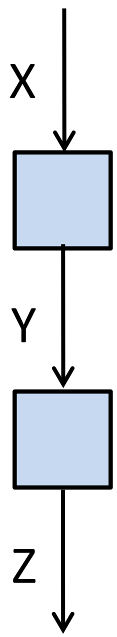
```
make_filter<void,Y>(
    filter::serial_in_order,
    [&]( flow_control& fc ) -> Y {
        Y y;
        cin >> y;
        if( cin.fail() ) fc.stop();
        return y;
    }
)
```

serial_out_of_order or **parallel** allowed too.

First stage receives special **flow_control** argument.

Composing Stages

- Compose stages with **operator&**




```
make_filter<X,Y>(
    ...
)
&
make_filter<Y,Z>(
   ...
)
```

types must match

Type Algebra

make_filter<*T*,*U*>(*mode*,*functor*) $\rightarrow$ filter_t<*T*,*U*>

filter_t<*T*,*U*> & filter_t<*U*,*V*> $\rightarrow$ filter_t<T,V>

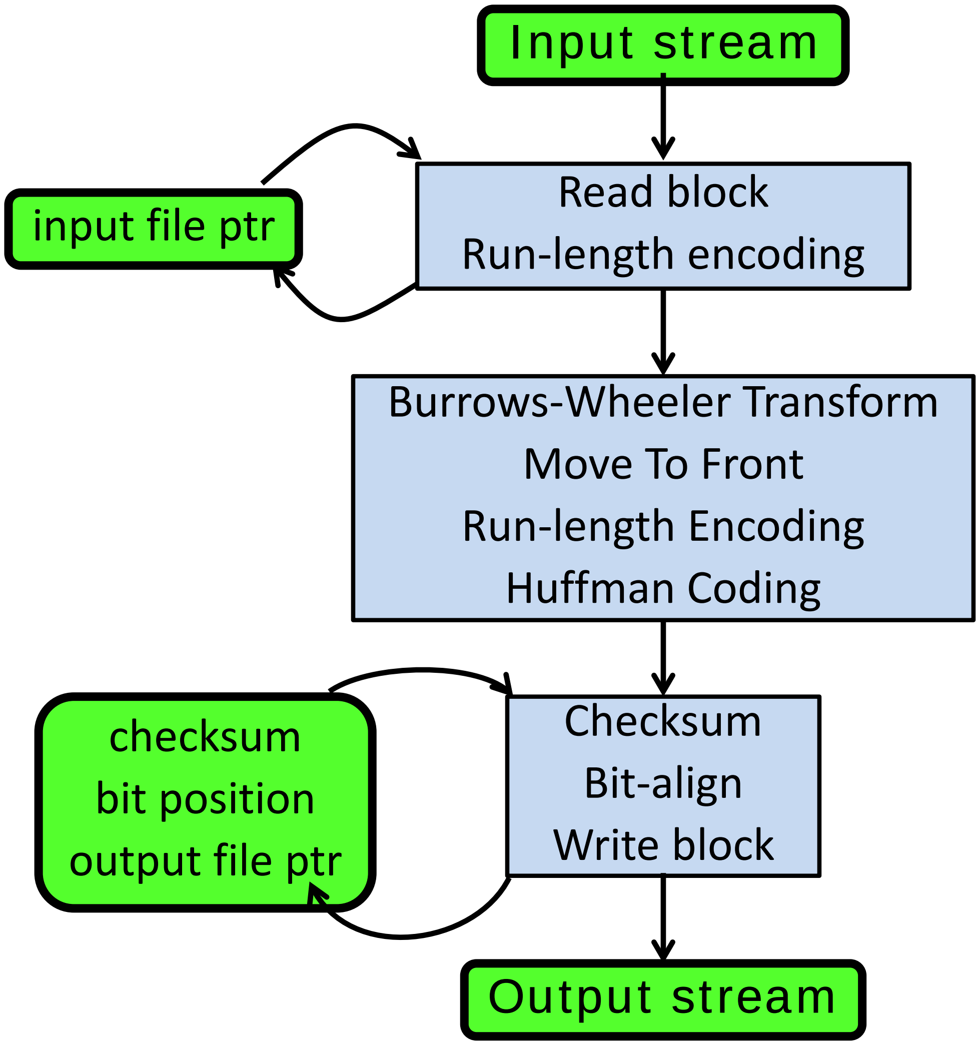
Running the Pipeline

parallel_pipeline(
    size_t *ntoken*, const filter_t<void,void>& *filter*);

Token limit.

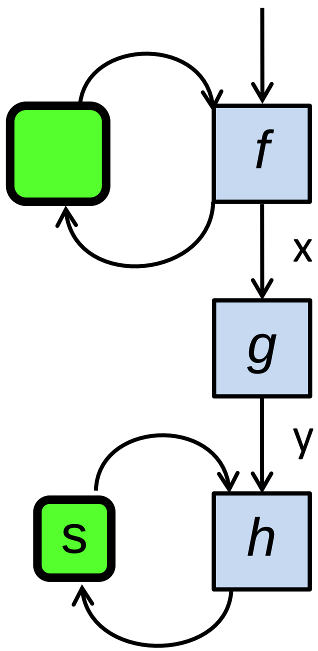
filter must map void→void.

Bzip2 with `parallel_pipeline`



Pipeline Hack in Cilk Plus

- General TBB-style pipeline not possible (yet)
- But 3-stage serial-parallel-serial is.



serial loop

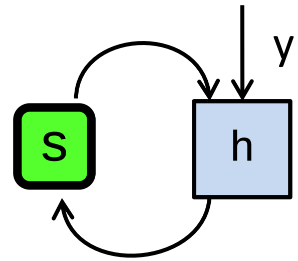
y = **cilk_spawn** *g*(x)

s = *h*(s,y)

Problem: *h* is *not* always associative

Monoid via Deferral

- Element is *state* or *list*
  - $list = \{y_0, y_1, \ldots y_n\}$
- Identity = {}
- Operation = $\otimes$
  - $list_1 \otimes list_2 \rightarrow list_1$ concat $list_2$
  - $state \otimes list \rightarrow state'$
  - $\ldots \otimes state_2$ disallowed



s = h(s,y)

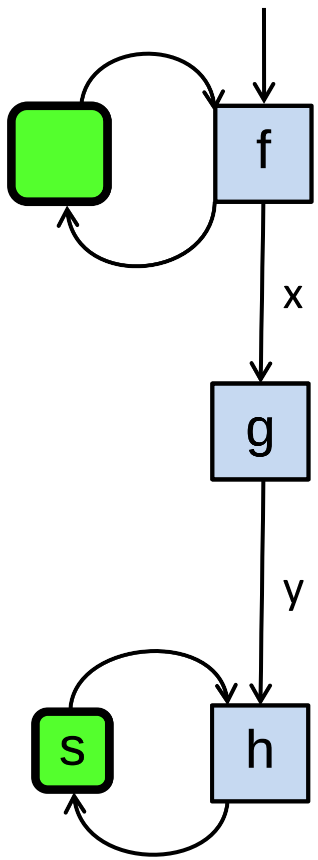
```
reducer_consume sink<S,Y>( &s, h );
```

Declare the stage

```
sink.consume(y);
```

Feed one item to it.

All Three Stages




```
while( T x = f() )
    cilk_spawn Stage2(x);
cilk_sync;
```

```
void Stage2( T x ) {
    sink.consume(g(x));
}
```

```
S s;
reducer_consume<S,U> sink (
    &s, h
);
```

Implementing reducer_consume

```
#include <cilk/reducer.h>
#include <list>

template<typename State, typename Item>
class reducer_consume {
public:
    typedef void (*consumer_func)(State*,Item);
private:
    struct View {...};
    struct Monoid: cilk::monoid_base<View> {...};
    cilk::reducer<Monoid> impl;
public:
    reducer_consume( State* s, consumer_func f );
};
```

view for a strand

our monoid

representation of reducer

reducer_consumer::View

```
struct View {
    std::list<Item> items;
    bool is_leftmost;
    View( bool is_leftmost_=false ) : is_leftmost(is_leftmost_) {}
    ~View() {}
};
```

reducer_consumer::Monoid

```
struct Monoid: cilk::monoid_base<View> {
    State* state;
    consumer_func func;
    void munch( const Item& item ) const {
        func(state,item);
    }
    void reduce(View* left, View* right) const {
        assert( !right->is_leftmost );
        if( left->is_leftmost )
            while( !right->items.empty() ) {
                munch(right->items.front());
                right->items.pop_front();
            }
        else
            left->items.splice( left->items.end(), right->items );
    }
    Monoid( State* s, consumer_func f ) : state(s), func(f) {}
};
```

default implementation for a monoid

define "left = left ⊗ right"

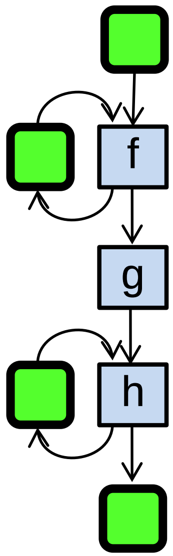
Finishing the Wrapper

```
template<typename State, typename Item>
class reducer_consume {
public:
   typedef void (*consumer_func)(State*,Item);
private:
   struct View;
   struct Monoid;
   cilk::reducer<Monoid> impl;
public:
   reducer_consume( State* s, consumer_func f) : impl(Monoid(s,f), /*is_leftmost=*/true) {}
   void consume( const Item& item ) {
      View& v = impl.view();
      if( v.is_leftmost )
         impl.monoid().munch( item );
      else
         v.items.push_back(item);
   }
};
```

Initial monoid

Argument to initial View

Pipeline Pattern



Intel® Cilk™ Plus(special case)

```
S s;
reducer_consume<S,U> sink (
   &s, h
);
...
void Stage2( T x ) {
   sink.consume(g(x));
}
...
while( T x = f() )
   cilk_spawn Stage2(x);
cilk_sync;
```

Thread parallelism

Intel® TBB

```
parallel_pipeline (
   ntoken,
   make_filter<void,T>(
      filter::serial_in_order,
      [&]( flow_control & fc ) -> T{
         T item = f();
         if( !item ) fc.stop();
         return item;
      }
   ) &
   make_filter<T,U>(
      filter::parallel,
      g
   ) &
   make_filter<U,void>(
      filter:: serial_in_order,
      h
   )
);
```

Summary(1)

- Cilk Plus and TBB are similar at an abstract level.
  - Both enable parallel pattern work-horses
    - Map, reduce, fork-join
- Details differ because of design assumptions
  - Cilk Plus is a language extension with compiler support.
  - TBB is a pure library approach.
  - Different syntaxes

Summary (2)

- Vector parallelism
  - Cilk Plus has two syntaxes for vector parallelism
    - Array Notation
    - **`#pragma simd`**
  - TBB does not support vector parallelism.
    - TBB + **`#pragma simd`** is an attractive combination
- Thread parallelism
  - Cilk Plus is a strict fork-join language
    - Straitjacket enables strong guarantees about space.
  - TBB permits arbitrary task graphs
    - "Flexibility provides hanging rope."

CONCLUSION

Intel® Cilk™ Plus	Intel® Threading Building Blocks	Domain Specific Libraries	Established Standards	Research and Development
C/C++ language extensions to simplify parallelism Open sourced Also an Intel product	Widely used C++ template library for parallelism Open sourced Also an Intel product	Intel® Integrated Performance Primitives Intel® Math Kernel Library	Message Passing Interface (MPI) OpenMP* Coarray Fortran OpenCL*	Intel® Concurrent Collections Offload Extensions Intel® Array Building Blocks River Trail: parallel javascript Intel® SPMD Parallel Compiler

Choice of high-performance parallel programming models

- Libraries for pre-optimized and parallelized functionality
- Intel® Cilk™ Plus and Intel® Threading Building Blocks supports composable parallelization of a wide variety of applications.
- OpenCL* addresses the needs of customers in specific segments, and provides developers an additional choice to maximize their app performance
- MPI supports distributed computation, combines with other models on nodes

Other Parallel Programming Models

OpenMP

- Syntax based on pragmas and an API
- Works with C, C++, and Fortran
- As of OpenMP 4.0 now includes explicit vectorization

MPI

- Distributed computation, API based

Co-array Fortran

- Distributed computation, language extension

OpenCL

- Uses separate kernel language plus a control API
- Two-level memory and task decomposition hierarchy

CnC

- Coordination language based on a graph model
- Actual computation must be written in C or C++

ISPC

- Based on elemental functions, type system for uniform computations

River Trail

- Data-parallel extension to Javascript

Course Summary

- Have presented a subset of the parallel programming models available from Intel
  - Useful for writing efficient and scalable parallel programs
  - Presented Cilk Plus and Threading Building Blocks (TBB)
- Also presented structured approach to parallel programming based on patterns
  - With examples for some of the most important patterns
- A book, *Structured Parallel Programming: Patterns for Efficient Computation*, is available that builds on the material in this course:

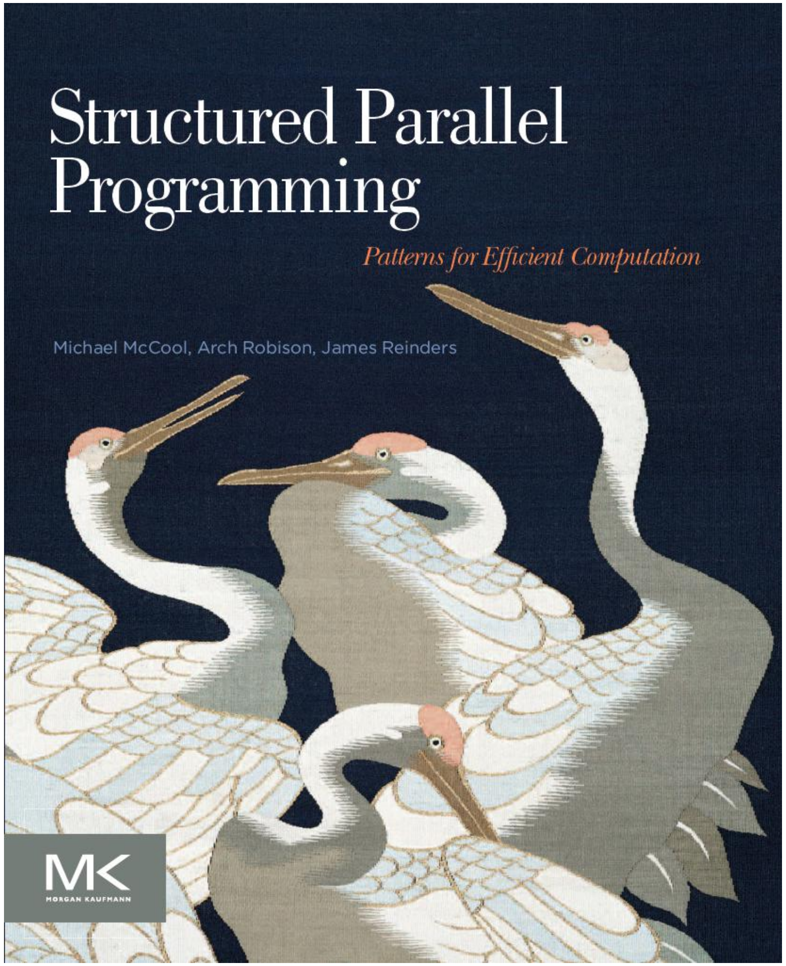
  http://parallelbook.com

For More Information

Structured Parallel Programming: Patterns for Efficient Computation

- Michael McCool
- Arch Robison
- James Reinders

- Uses Cilk Plus and TBB as primary frameworks for examples.
- Appendices concisely summarize Cilk Plus and TBB.
- www.parallelbook.com



intel®

Software

Optimization Notice

Optimization Notice

Intel's compilers 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 SSE2®, SSE3, and 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. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Legal Disclaimer

http://intel.com/software/products