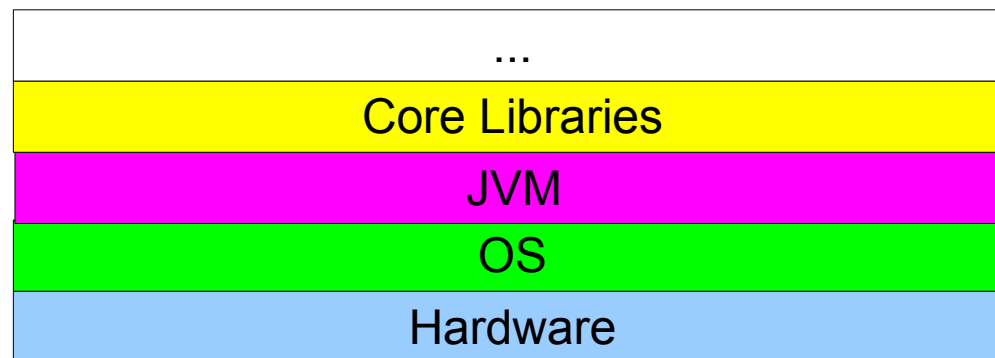

Supporting the many flavors of parallel programming

Doug Lea
SUNY Oswego

Outline

◆ Parallelism

- ◆ Transition from hidden implementation techniques to supporting explicit parallel/concurrent programming
 - ◆ Most techniques that make systems faster involve parallelism
 - ◆ Within and across layers



◆ Diversity

- ◆ Ideas behind support for some parallel FP, OO, and ADT based styles, patterns, and idioms

Parallelism in Implementations

- ◆ **Over forty years of parallelism and asynchrony in implementations for commodity platforms**
 - ◆ **Overlapped IO and device control**
 - ◆ **OS Processes and scheduling**
 - ◆ **Networked/distributed system handlers/modules**
 - ◆ **Event/GUI handlers; interrupts**
 - ◆ **Superscalar, out-of-order ALUs**
 - ◆ **Custom co-processors, ASICs, GPUs etc**
 - ◆ **Concurrent garbage collection and VM services**
- ◆ **Result in better throughput and/or latency**
 - ◆ **But point-wise, quirky; no grand plan**
 - ◆ **Complex performance models**

Exposing Parallelism

Old Elitism: Hide from most programmers

- ◆ “Programmers think sequentially”
- ◆ “Only an expert should try to write a <X>”
- ◆ “<Y> is a kewl hack but too weird to export”

End of an Era

- ◆ Few remaining hide-able speedups (Amdahls law)
- ◆ Hiding is impossible with multicores, GPUs, FPGAs

New Populism: Embrace and rationalize

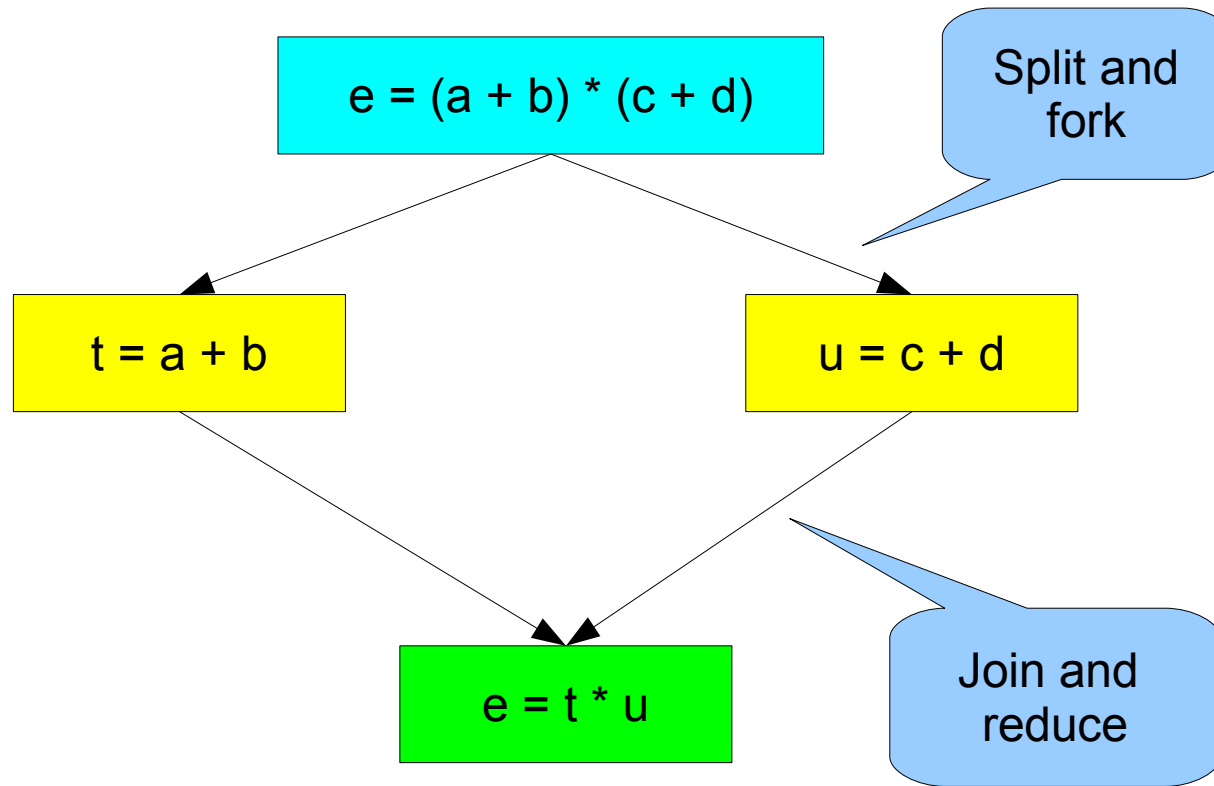
- ◆ Must integrate with defensible programming models, language support, and APIs
- ◆ Some residual quiriness is inevitable

Diversity

Parallel and concurrent programming have many roots

- ◆ **Functional, Object-oriented, and ADT-based procedural patterns are all well-represented; including:**
 - ◆ **Parallel (function) evaluation**
 - ◆ **Bulk operations on aggregates (map, reduce etc)**
 - ◆ **Shared resources (shared registries, transactions)**
 - ◆ **Sending messages and events among objects**
- ◆ **But none map uniformly to platforms**
 - ◆ **Beliefs that any are most fundamental are delusional**
 - ◆ **Arguments that any are “best” are silly**

Parallel Evaluation

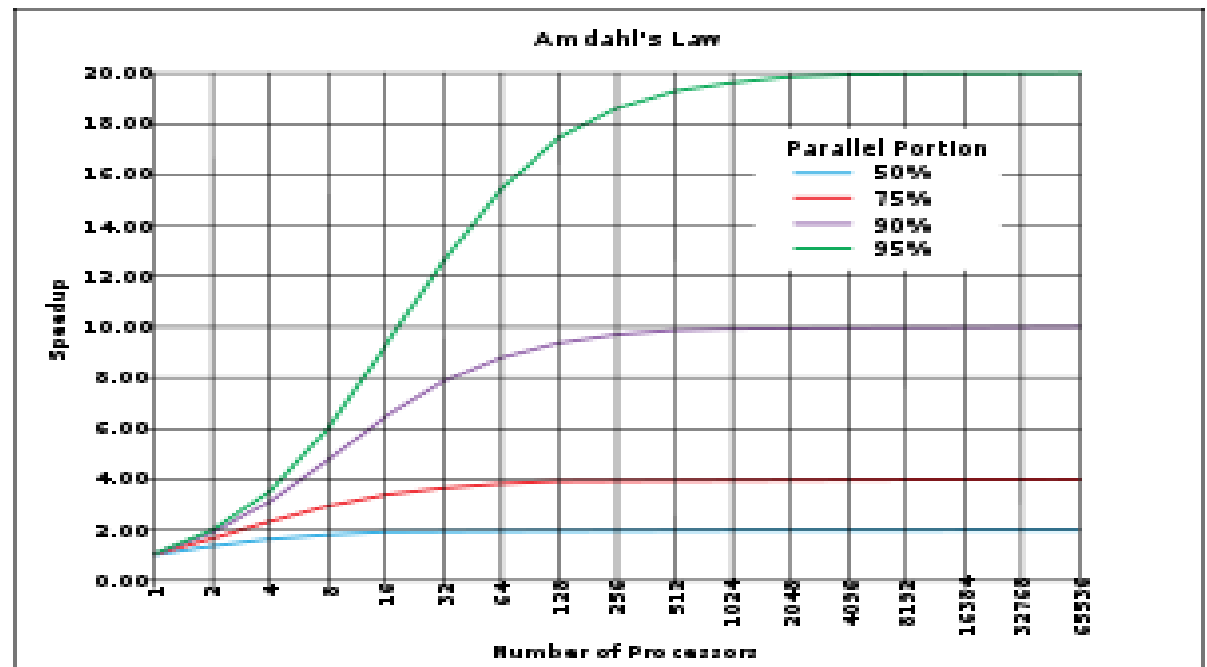


Parallel divide and conquer

Limits of Parallel Evaluation

- ◆ Why can't we always parallelize to turn any $O(N)$ problem into $O(N / \text{\#processors})$?
 - ◆ Sequential dependencies and resource bottlenecks
- ◆ For program with serial time S , and parallelizable fraction f , max speedup regardless of \#proc is $1 / ((1 - f) + f / S)$
- ◆ Can also express in terms of critical paths or tree depths

Wikipedia



Parallel Evaluation inside CPUs

Goal: Parallelize basic expressions

- ◆ Problem: Instructions are in sequential stream

Idea: Dependency-based execution

- ◆ Complete instructions when inputs are ready (from memory reads or ops) and outputs are available
 - ◆ Uses a hardware-based variant of dataflow analysis
- ◆ Pipelines, buffered in-flight instructions, out-of-order processing, multiple ALUs, store buffers, etc

Problem: It is always on!

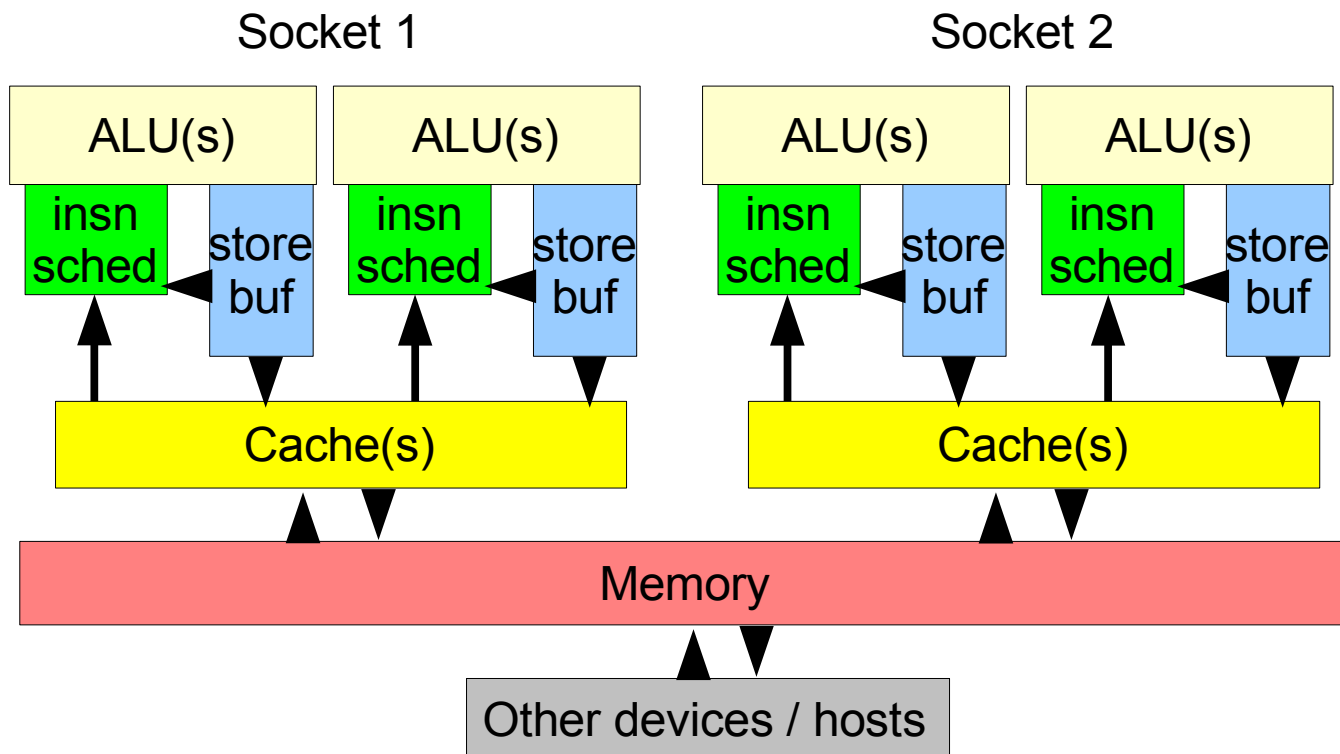
- ◆ Dependency analysis is shallow, local, and may not match (concurrent) program semantics
- ◆ Undefined in presence of races: what if another processor modifies a variable accessed in an instruction?

Parallelizing Arbitrary Expressions

- ◆ **Instruction-level parallelism doesn't scale well**
 - ◆ But can use similar ideas on multicores
 - ◆ With similar benefits and issues
- ◆ **Example: `val e = f(a, b) op g(c, d)`**
- ◆ **Easiest if rely on shallow dependency analysis**
 - ◆ Methods `f` and `g` are pure, independent functions
 - ◆ Can exploit commutativity and/or associativity
- ◆ **Other cases require harder work**
 - ◆ To find smaller-granularity independence properties
 - ◆ For example, parallel sorting, graph algorithms
 - ◆ Harder work → more bugs; sometimes more payoff

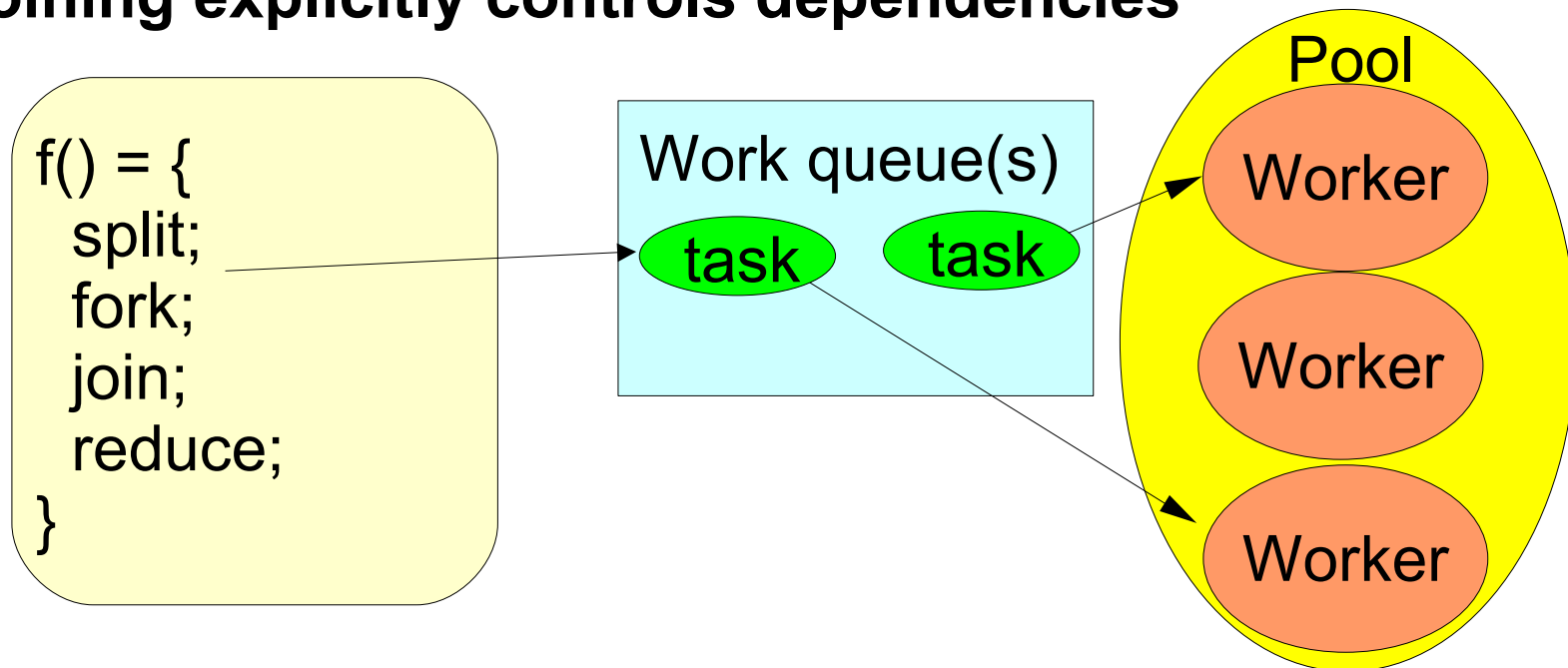
Multiprocessors and Multicores

- ◆ At least two new layers of parallelism
 - ◆ But coarser-grained
 - ◆ Split by instruction streams (threads)



Task-Based Parallel Evaluation

- ◆ Programs can be broken into tasks
 - ◆ Recursively; under some appropriate level of granularity
- ◆ Workers/Cores continually run tasks
 - ◆ Sub-computations are forked as subtask objects
- ◆ Sometimes need to wait for subtasks
 - ◆ Joining explicitly controls dependencies

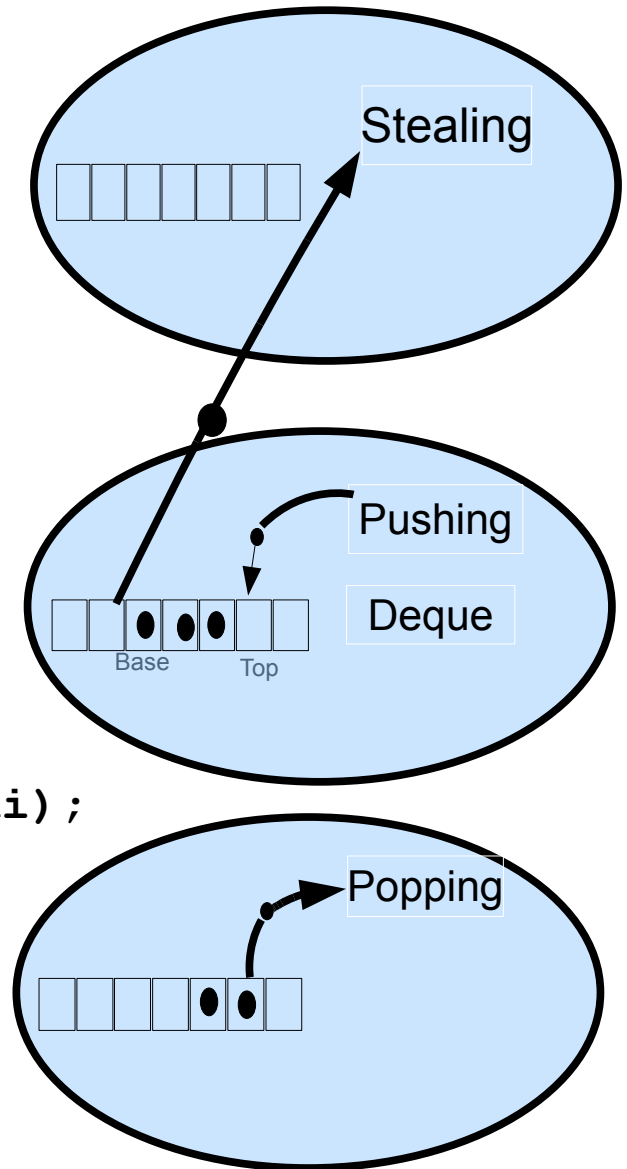


ForkJoin Sort (Java)

```
class SortTask extends RecursiveAction {
    final long[] array;
    final int lo; final int hi;

    SortTask(long[] array, int lo, int hi) {
        this.array = array;
        this.lo = lo; this.hi = hi;
    }

    protected void compute() {
        if (hi - lo < THRESHOLD)
            sequentiallySort(array, lo, hi);
        else {
            int m = (lo + hi) >>> 1;
            SortTask r = new SortTask(array, m, hi);
            r.fork();
            new SortTask(array, lo, m).compute();
            r.join();
            merge(array, lo, mid, hi);
        }
    }
    // ...
} // (In Java, since no standard Scala APIs yet.)
```



Implementing ForkJoin Tasks

Queuing: Work-stealing

- ◆ Each worker forks to own deque; but steals from others or accepts new submission when no work

Scheduling: Locally LIFO, random-steals FIFO

- ◆ Cilk-style: Optimal for divide-and-conquer
- ◆ Ignores locality: Cannot tell if better to use another core on same processor, or a different processor

Joining: Helping and/or pseudo-continuations

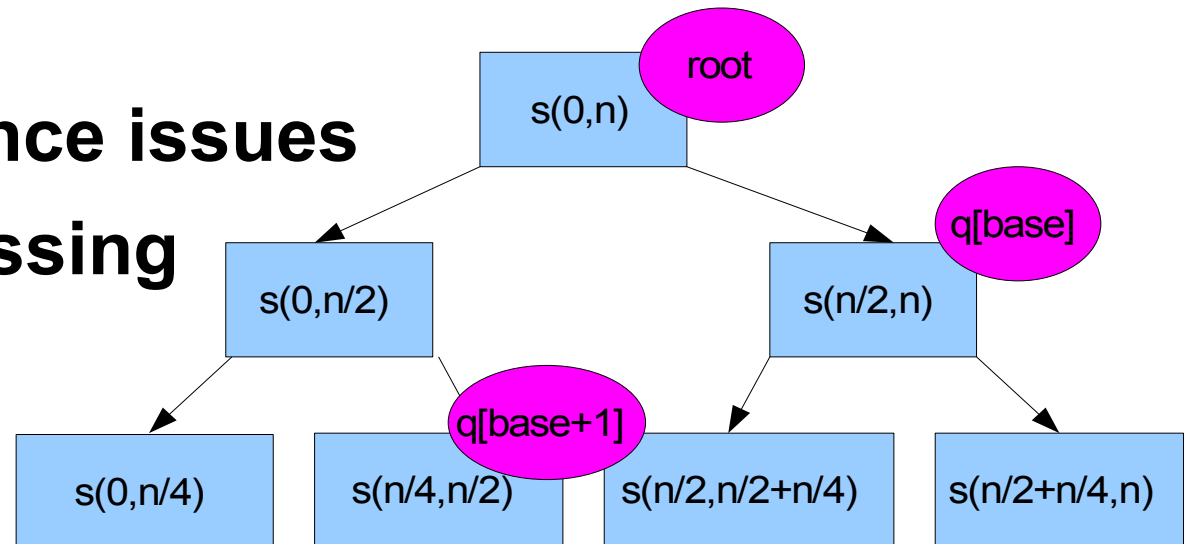
- ◆ Try to steal a child of stolen task; if none, block but (re)start a spare thread to maintain parallelism

Overhead: Task object with one 32-bit int status

- ◆ Payoff after ~100-1000 instructions per task body

Bulk Operations

- ◆ **SIMD: Apply an operation to all elements of a collection**
 - ◆ Procedures: Color all my squares red
 - ◆ Mappings: Map these student IDs to their grades
 - ◆ Reductions: Calculate the sum of these numbers
- ◆ **A special case of basic parallel evaluation**
 - ◆ Any number of components; same operation on each
 - ◆ Same independence issues
- ◆ **Can arrange processing in task trees/dags**
 - ◆ **Array Sum:**



Supporting Bulk Operations

- ◆ **Familiar APIs and usages: map, reduce/fold, filter,...**
 - ◆ **Common in many styles of programming**
 - ◆ Requires use of closures to express ops
 - ◆ (See other talks for Scala details)
 - ◆ **But some unfamiliar twists**
 - ◆ Avoid sequential iterators, head::tail, etc
 - ◆ **Functional style more readily optimizable**
 - ◆ Ex: `filter(p1).filter(p2) → filter(p1 & p2)`
- ◆ **Many ways to implement**
 - ◆ **Possibly via: SIMD instructions, GPUs, FPGAs, ForkJoin, Clusters (Hadoop etc)**
 - ◆ But not even close to automating selection yet.

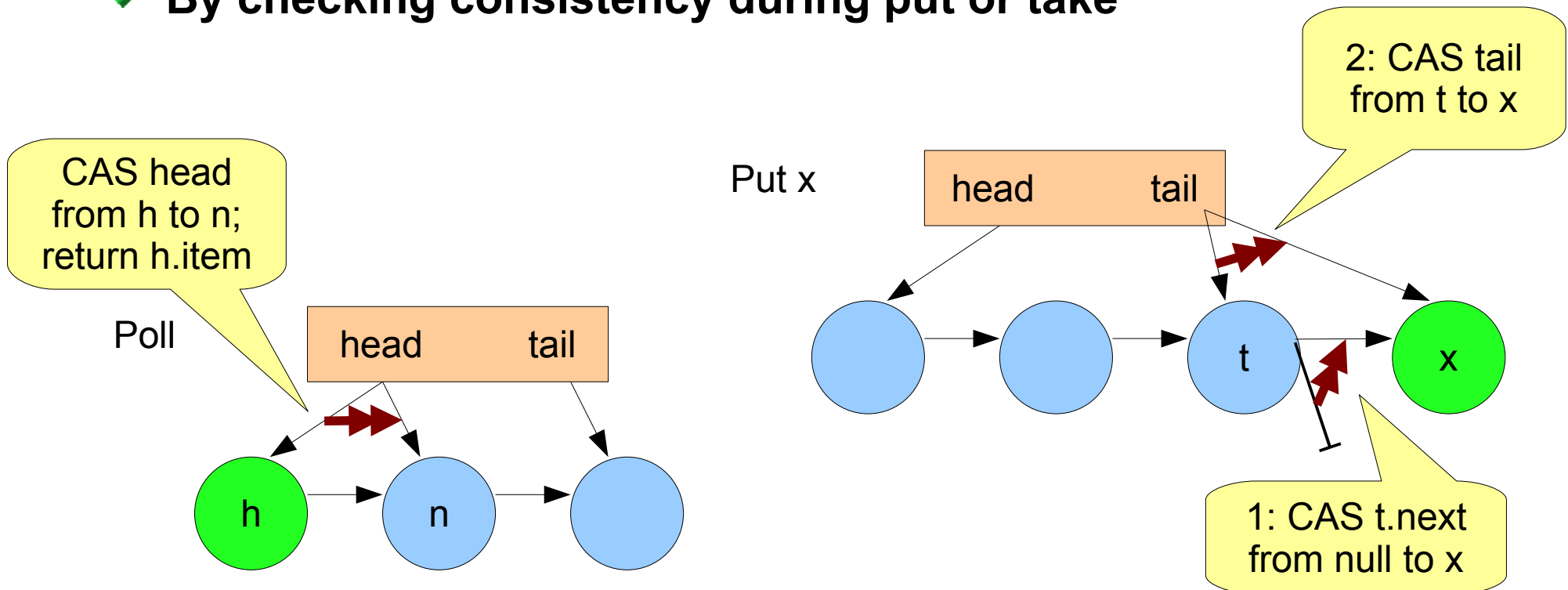
Semi-Transactional ADTs

- ◆ **Explicitly concurrent objects used as resources**
 - ◆ Support conventional APIs (Collections, Maps)
 - ◆ Examples: Registries, directories, message queues
 - ◆ Programmed in low-level JVMese – compareAndSet (CAS)
 - ◆ Often vastly more efficient than alternatives
- ◆ **Roots in ADTs and Transactions**
 - ◆ ADT: Opaque, self-contained, limited extensibility
 - ◆ Transactional: All-or-nothing methods
- ◆ **Atomicity limits; e.g., no transactional removeAll**
 - ◆ But can support *non-transactional* bulk *parallel* ops
 - ◆ (Need for transactional parallel bulk ops is unclear)
 - ◆ Possibly only transiently concurrent
 - ◆ Example: Shared outputs for bulk parallel operations

Example: Non-blocking Queues

Michael & Scott Queue (PODC 1996)

- Use retriabale CAS (not lock)
- CASes on different vars (head, tail) for put vs poll
- If CAS of tail from t to x on put fails, others try to help
 - By checking consistency during put or take



Concurrent Collections

- ◆ **Non-blocking data structures rely on simplest form of hardware transactions**
 - ◆ **CAS (or LL/SC) tries to commit a single variable**
 - ◆ **Frameworks layered on CAS-based data structures can be used to support larger-grained transactions**
 - ◆ **HTM (or multiple-variable CAS) would be nicer**
 - ◆ **But not a magic bullet**
- ◆ **Evade most hard issues in general transactions**
 - ◆ **Contention, overhead, space bloat, side-effect rollback, etc**
 - ◆ **But special cases of these issues still present**
 - ◆ **Complicates implementation: Hard to see Michael & Scott algorithm hiding in `LinkedTransferQueue`**

Contention in Shared Data Structures

Mostly-Write

- ◆ Most producer-consumer exchanges
 - ◆ Especially queues
- ◆ Apply combinations of a small set of ideas
 - ◆ Use non-blocking sync via compareAndSet (CAS)
 - ◆ Reduce point-wise contention
 - ◆ Arrange that threads help each other make progress

Mostly-Read

- ◆ Most Maps & Sets
 - ◆ Empirically, 85% Java Map calls read-only
- ◆ Structure to maximize concurrent readability
 - ◆ Without locking, readers see legal (ideally, linearizable) values
 - ◆ Often, using immutable copy-on-write internals
 - ◆ Apply write-contention techniques from there

Objects, Actors, Messages, Events

- ◆ **Reactive GUI, Web, embedded, etc applications**
 - ◆ Almost necessarily object-oriented
- ◆ **Many choices for semantics**
 - ◆ Allow both actors and passive objects?
 - ◆ One actor (aka, the event loop) vs many?
 - ◆ Single- vs multi- threaded vs transactional actors?
 - ◆ Isolated (process-like) vs shared memory?
 - ◆ Explicitly remote vs local actors?
 - ◆ Point-to-point messaging vs multicast events?
 - ◆ Synchronous vs asynchronous messaging?
 - ◆ Support exceptions and Faults?
- ◆ **JVM and libraries supply mechanism, not policy**

Common Infrastructure

◆ Lightweight Actors

- ◆ Similar (and can be identical) to lightweight Tasks
- ◆ Multiplex to threads/processes/cores/hosts
 - ◆ Normally via Executor API

◆ Shared-memory sync support

- ◆ Queues, Futures, Locks, Barriers, etc
- ◆ Shared is faster than unshared messaging
 - ◆ But can be less scalable for point-to-point
- ◆ Provides stronger guarantees: Cache coherence
- ◆ Can be more error-prone: Aliasing, races, visibility
- ◆ Exposing benefits vs complexity is policy issue

Events and Consistency

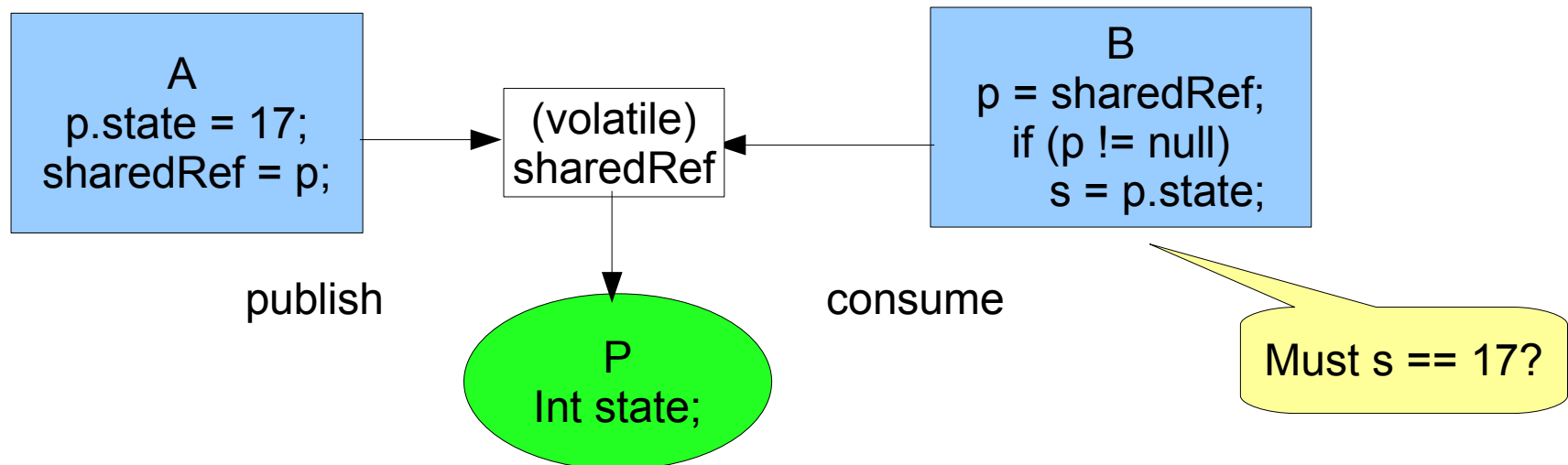
- ◆ **Consistency issues are intrinsic to event systems**
 - ◆ **Example: vars x,y initially 0 → events x, y unseen**
 - ◆ **Node A: send x = 1; // (multicast send)**
 - ◆ **Node B: send y = 1;**
 - ◆ **Node C: receive x; receive y; // see x=1, y=0**
 - ◆ **Node D: receive y; receive x; // see y=1, x=0**
- ◆ **On shared memory, can guarantee agreement**
 - ◆ **JMM: declare x, y as volatile**
- ◆ **Remote consistency is expensive**
 - ◆ **Atomic multicast, distributed transactions; failure models**
- ◆ **Usually, weaker consistency is good enough**
 - ◆ **Example: Per-producer FIFO**

Memory Models

- ◆ Distinguish **sync** accesses (locks, volatiles, atomics) from **normal** accesses (reads, writes)
- ◆ Require strong ordering properties among **sync**
 - ◆ Usually “strong” means Sequentially Consistent
- ◆ Allow as-if-sequential reorderings among **normal**
 - ◆ Usually means: obey seq data/control dependencies
- ◆ Restrict reorderings between **sync** vs **normal**
 - ◆ Rules usually not obvious or intuitive
 - ◆ Special rules for cases like final fields
- ◆ There's probably a better way to go about all this

Example: Ownership Transfer

- ◆ When B gains access to object P, it expects to see the state of P left by A that provided access.
- ◆ Concurrent languages provide some means to express this (e.g., Java volatiles and atomics)
- ◆ All j.u.c components guarantee safe transfer when applicable, so users never need to think about it.



Conclusions

◆ Parallelism is everywhere

- ◆ Can be natural and elegant
- ◆ Can be mind-numbingly messy and complex
- ◆ Usually somewhere in-between
- ◆ Just like every other aspect of programming

◆ Diversity is essential

- ◆ Functional, OO, ADT styles all apply often
- ◆ Language/library support for each continue to evolve
 - ◆ Encapsulate messiness; reduce complexity
- ◆ Just like every other aspect of programming

Josh Bloch's version of this slide:
“Life's a bitch, but the puppies are cute”